Quinta edición

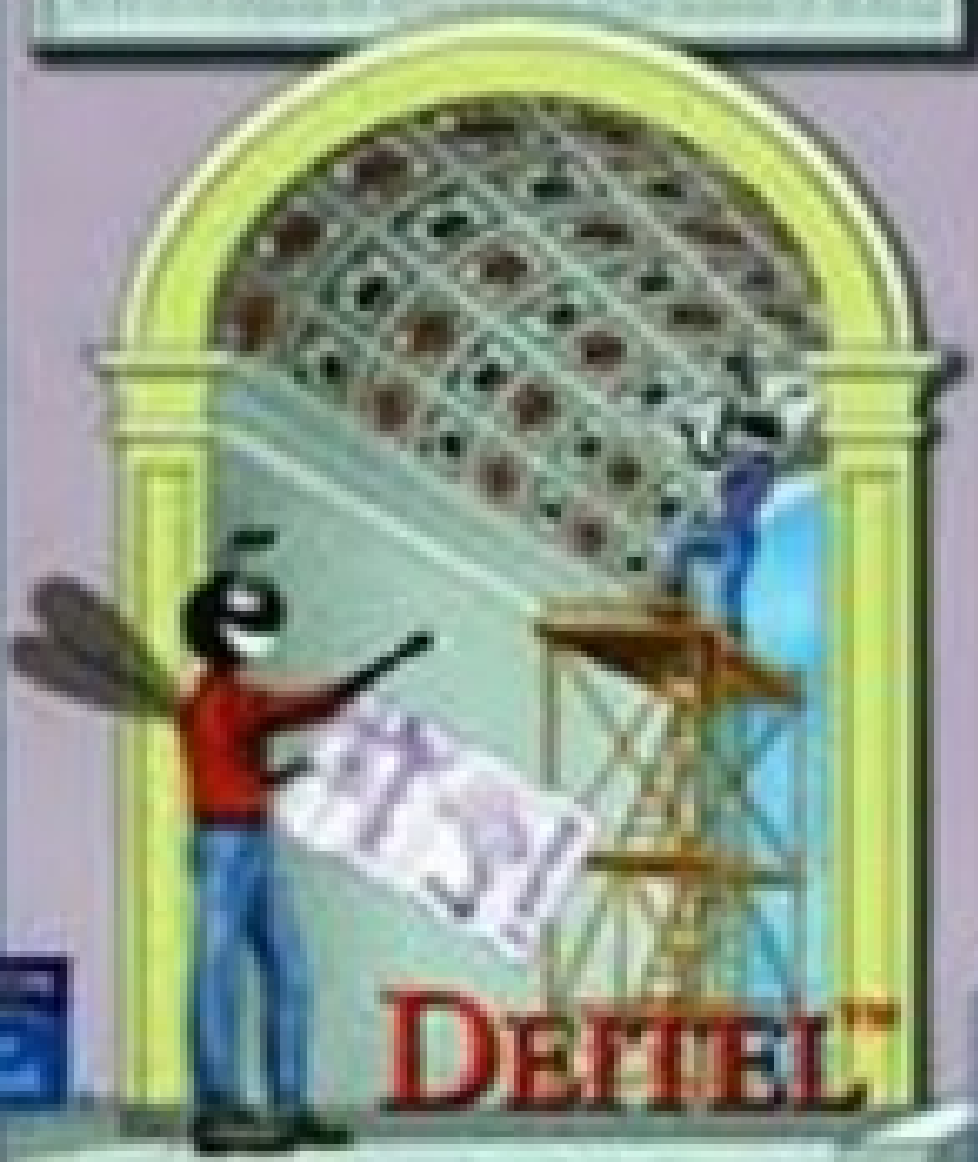
JAVA™ CÓMO PROGRAMAR

Introducción al

DOO con UML y los

PATRONES DE DISEÑO

JDBC™, SERVLETS, JSP™



DEITEL™

DEITEL
DEITEL

Quinta edición

JAVA™ CÓMO PROGRAMAR

DEITEL • DEITEL

Java™ ha revolucionado el desarrollo del software mediante el código orientado a objetos independiente de la plataforma, con uso intensivo de multimedia, para aplicaciones basadas en Internet, intranets o extranets. Esta *quinta edición* del libro de texto sobre Java más utilizado a nivel mundial, explica cómo utilizar las extraordinarias herramientas de este software.

La obra contiene una vasta colección de ejemplos, ejercicios y proyectos que proporcionan la oportunidad de resolver problemas reales. El contenido se concentra en los principios de la buena ingeniería de software, haciendo hincapié en la claridad de los programas. Todos los ejemplos de código han sido probados.
Entre los temas clave que se tratan aquí, se encuentran:

- Aplicaciones/Applets
- GUI Swing/Manejo de eventos
- Clases/Objetos/Interfaces
- POO/Herencia/Polimorfismo
- Gráficos/Java 2D™/Imágenes/Animación/Audio
- Excepciones/Subprocesamiento múltiple
- DOO/UML/Patrones de diseño

Cómo programar en Java, quinta edición incluye extensas características pedagógicas:

- Cientos de programas en código activo (LIVE-CODE™) con capturas de pantallas que muestran los resultados obtenidos
- Una gran cantidad de recursos en Internet y Web para impulsar al lector a que investigue más sobre los temas de su interés
- Cientos de tips, prácticas recomendadas y precauciones. Todos identificados con su respectivo icono:



En el sitio Web `www.pearsoneducacion.net/deitel` encontrará los ejemplos de código del libro e información para maestros, estudiantes y profesionales.
Sin duda, éste es el libro más completo para aprender Java.

El CD incluye:

- Archivos de texto en *español* no incluidos en el libro
- Todo el código fuente utilizado en los ejercicios del libro
- Programas para ejecutar Java, como Java™ 2 Platform Software Development Kit y Apache Tomcat v.4.1.12, y mucho más

ISBN 970-26-0518-0

Visítenos en:
www.pearsoneducacion.net

DEITEL

JAVA™ CÓMO PROGRAMAR

Quinta edición

Introducción al DOO con UML y los PATRONES DE DISEÑO JDBC™, SERVLETS, JSP™

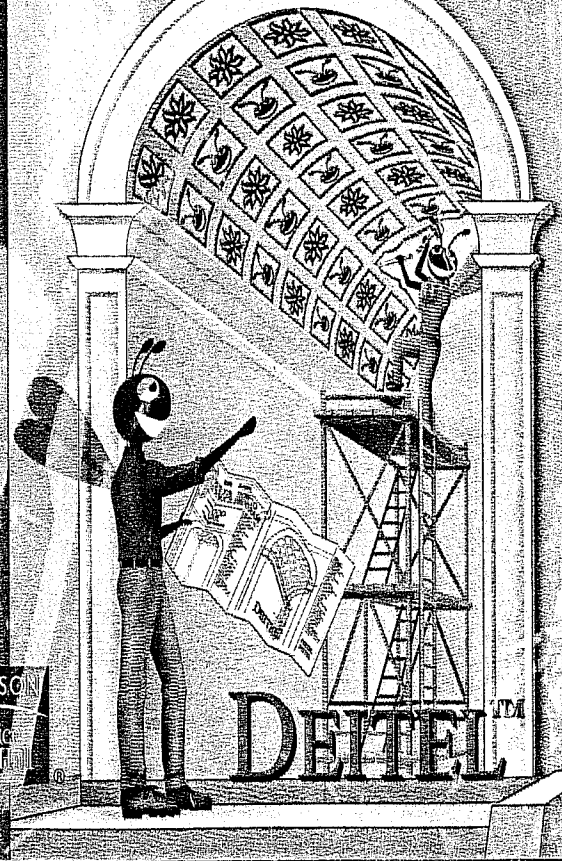
INCLUYE CD-ROM

Quinta edición

Incluye CD con el Kit de desarrollo de software Java™ 2, Standard Edition, Apache Tomcat v.4.1.12, ¡y mucho más!

JAVA™ CÓMO PROGRAMAR

Introducción al DOO con UML y los PATRONES DE DISEÑO JDBC™, SERVLETS, JSP™



DEITEL

DEITEL

http://librosysolucionarios.net

QUINTA EDICIÓN

CÓMO PROGRAMAR EN JAVA

http://librosysolucionarios.net

QUINTA EDICIÓN

CÓMO PROGRAMAR EN JAVA

Harvey M. Deitel
Deitel & Associates, Inc.

Paul J. Deitel
Deitel & Associates, Inc.

TRADUCCIÓN

Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

REVISIÓN TÉCNICA

M. en C. Gabriela Azucena Campos García
Profesora de tiempo completo
Departamento de Sistemas de Información
División de Profesional y Graduados
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México

M. en C. Sergio Fuenlabrada Velázquez
Ing. Mario Alberto Sesma Martínez
M.C.I. Edna Martha Miranda Chávez
Profesores investigadores
Academia de computación
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

M. en C. Jesús Muñoz Bauza
Profesor investigador
Departamento de Computación
División de Profesional y Graduados
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

PEARSON Educación®

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

http://librosysolucionarios.net

Datos de catalogación bibliográfica

DEITEL, HARVEY M. y DEITEL, PAUL J.
Quinta edición

Cómo programar en Java

PEARSON EDUCACIÓN, México, 2004

ISBN: 970-26-0518-0
Área: Universitarios

Formato: 20 × 25.5 cm Páginas: 1268

Authorized translation from the English language edition, entitled *Java How to Program, Fifth Edition*, by *Harvey M. Deitel* and *Paul J. Deitel*, published by Pearson Education, Inc., publishing as PRENTICE HALL, INC., Copyright ©2003. All rights reserved.

ISBN 0-13-101621-0

Traducción autorizada de la edición en idioma inglés, titulada *Java How to Program, Fifth Edition*, por *Harvey M. Deitel* y *Paul J. Deitel*, publicada por Pearson Education, Inc., publicada como PRENTICE HALL INC., Copyright ©2003. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español:
Editor: Guillermo Trujano Mendoza
e-mail: guillermo.trujano@pearsoned.com
Editor de desarrollo: Miguel B. Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

Edición en inglés:
Vice President and Editorial Director, ECS: ***Marcia J. Horton***
Acquisitions Editor: ***Petra J. Recter***
Assistant Editor: ***Sarah Parker***
Project Manager: ***Jennifer Cappello***
Vice President and Director of Production and Manufacturing, ESM: ***David W. Riccardi***
Executive Managing Editor: ***Vince O'Brien***
Managing Editor: ***Tom Manshreck***
Production Editor: ***John F. Lovell***
Director of Creative Services: ***Paul Belfanti***
Creative Director: ***Carole Anson***
Chapter Opener and Cover Designer: ***Tamara L. Newnam, Dr. Harvey Deitel***
Interior Design Assistance: ***Geoffrey Cassar***
Manufacturing Manager: ***Trudy Pisciotti***
Manufacturing Buyer: ***Lisa McDowell***
Marketing Manager: ***Pamela Shaffer***
Marketing Assistant: ***Barrie Reinhold***

QUINTA EDICIÓN, 2004

D.R. © 2004 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México
e-mail: editorial.universidades@pearsoned.com

Cámara Nacional de la Industria Editorial Mexicana.
Reg. Núm. 1031.

Prentice Hall es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN 970-26-0518-0

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 05 04 03 02

http://librosysolucionarios.net

PARA:

Terell Hull y James Huddleston:

Por su constante dedicación a la excelencia en la enseñanza y al escribir sobre Java y la tecnología de objetos.

Por sus extraordinarias contribuciones como revisores y por insistir en que "lo hiciéramos bien".

Gracias por ser nuestros mentores, colegas y amigos.

Es un privilegio trabajar con profesionales consumados en el software.

Harvey y Paul Deitel

Contenido

	Prefacio	**xvii**
1	**Introducción a las computadoras, Internet y Web**	**1**
1.1	Introducción	2
1.2	¿Qué es una computadora?	4
1.3	Organización de las computadoras	4
1.4	Evolución de los sistemas operativos	5
1.5	Computación personal, distribuida y cliente/servidor	6
1.6	Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel	6
1.7	Historia de C++	7
1.8	Historia de Java	8
1.9	Bibliotecas de clases de Java	8
1.10	FORTRAN, COBOL, Pascal y Ada	9
1.11	BASIC, Visual Basic, Visual C++, C# y .NET	10
1.12	Internet y World Wide Web	11
1.13	Fundamentos de un entorno típico en Java	11
1.14	Generalidades acerca de Java y este libro	14
1.15	Acerca de los objetos: Introducción a la tecnología de objetos y el Lenguaje Unificado de Modelado	16
1.16	Descubrimiento de patrones de diseño: Introducción	19
2	**Introducción a las aplicaciones en Java**	**27**
2.1	Introducción	28
2.2	Su primer programa en Java: Imprimir una línea de texto	28
2.3	Modificación de nuestro primer programa en Java	34
2.4	Cómo mostrar texto en un cuadro de diálogo	36
2.5	Otra aplicación en Java: Suma de enteros	40
2.6	Conceptos acerca de la memoria	45
2.7	Aritmética	46
2.8	Toma de decisiones: Operadores de igualdad y relacionales	49
2.9	(Ejemplo práctico opcional) Acerca de los objetos: Cómo examinar el enunciado de un problema	55

http://librosysolucionarios.net

3 Introducción a los applets de Java 71
3.1 Introducción 72
3.2 Applets de muestra incluidos en el Kit de desarrollo de software para Java 2 72
3.3 Applet simple en Java: Cómo dibujar una cadena 78
3.4 Cómo dibujar cadenas y líneas 85
3.5 Cómo agregar números de punto flotante 87
3.6 Recursos en Internet y Web relacionados con los applets de Java 93
3.7 (Ejemplo práctico opcional) Acerca de los objetos: Identificación de las clases en el enunciado de un problema 94

4 Instrucciones de control: Parte 1 103
4.1 Introducción 104
4.2 Algoritmos 104
4.3 Seudocódigo 105
4.4 Estructuras de control 105
4.5 Estructura de selección `if` simple 108
4.6 Estructura de selección *`if...else`* 109
4.7 Estructura de repetición `while` 113
4.8 Cómo formular algoritmos: Ejemplo práctico 1 (repetición controlada por un contador) 114
4.9 Cómo formular algoritmos mediante el mejoramiento de arriba a abajo, paso a paso: Ejemplo práctico 2 (repetición controlada por un centinela) 118
4.10 Cómo formular algoritmos mediante el mejoramiento de arriba a abajo, paso a paso: Ejemplo práctico 3 (estructuras de control anidadas) 125
4.11 Operadores de asignación compuestos 129
4.12 Operadores de incremento y decremento 129
4.13 Tipos primitivos 132
4.14 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar los atributos de las clases 133

5 Instrucciones de control: Parte 2 145
5.1 Introducción 146
5.2 Fundamentos de la repetición controlada por contador 146
5.3 Instrucción de repetición `for` 148
5.4 Ejemplos sobre el uso de la instrucción `for` 151
5.5 Instrucción de repetición `do...while` 156
5.6 Instrucción de selección múltiple `switch` 158
5.7 Instrucciones `break` y `continue` 162
5.8 Instrucciones `break` y `continue` etiquetadas 164
5.9 Operadores lógicos 167
5.10 Resumen sobre programación estructurada 172
5.11 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar los estados y actividades de los objetos 177

6 Métodos 187
6.1 Introducción 188
6.2 Módulos de programas en Java 188
6.3 Métodos de la clase `Math` 189
6.4 Declaraciones de métodos 190
6.5 Promoción de argumentos 197
6.6 Paquetes de la API de Java 198
6.7 Generación de números aleatorios 198

6.8 Ejemplo: Un juego de azar 203
6.9 Alcance de las declaraciones 211
6.10 Métodos de la clase `JApplet` 214
6.11 Sobrecarga de métodos 215
6.12 Recursividad 217
6.13 Ejemplo del uso de la recursividad: La serie de Fibonacci 220
6.14 Comparación entre recursividad e iteración 225
6.15 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar las operaciones de las clases 226

7 Arreglos 245
7.1 Introducción 246
7.2 Arreglos 246
7.3 Declaración y creación de arreglos 247
7.4 Ejemplos acerca del uso de arreglos 248
7.5 Referencias y parámetros de referencia 258
7.6 Cómo pasar arreglos a los métodos 258
7.7 Ordenamiento de arreglos 261
7.8 Búsqueda en arreglos: Búsqueda lineal y búsqueda binaria 263
7.9 Arreglos multidimensionales 270
7.10 (Ejemplo práctico opcional) Acerca de los objetos: Colaboración entre los objetos 277

8 Programación basada en objetos 301
8.1 Introducción 302
8.2 Implementación de un tipo de dato abstracto con una clase 303
8.3 Alcance de las clases 310
8.4 Control del acceso a los miembros 310
8.5 Referencias a los miembros del objeto actual mediante `this` 311
8.6 Inicialización de los objetos de una clase: Constructores 313
8.7 Uso de constructores sobrecargados 314
8.8 Uso de los métodos *set (establecer)* y *get (obtener)* 318
8.9 Composición 326
8.10 Recolección de basura 329
8.11 Miembros de clase estáticos 330
8.12 Variables de instancia finales 334
8.13 Creación de paquetes 336
8.14 Acceso a los paquetes 341
8.15 Reutilización de software 343
8.16 Abstracción de datos y encapsulamiento 344
8.17 (Ejemplo práctico opcional) Acerca de los objetos: Cómo empezar a programar las clases para la simulación del elevador 345

9 Programación orientada a objetos: Herencia 355
9.1 Introducción 356
9.2 Superclases y subclases 357
9.3 Miembros `protected` 358
9.4 Relación entre las superclases y las subclases 359
9.5 Ejemplo práctico: Jerarquía de herencia de tres niveles 377
9.6 Constructores y finalizadores en las subclases 380
9.7 Ingeniería de software mediante la herencia 385

http://librosysolucionarios.net

10 Programación orientada a objetos: Polimorfismo 389
10.1 Introducción 390
10.2 Relaciones entre los objetos en una jerarquía de herencia 391
10.2.1 Invocación de los métodos de superclases desde objetos de subclases 392
10.2.2 Uso de referencias a superclases con variables tipo subclase 393
10.2.3 Llamadas a métodos de subclases mediante variables tipo superclase 394
10.3 Ejemplos de polimorfismo 396
10.4 Clases y métodos abstractos 397
10.5 Ejemplo práctico: Herencia de interfaz y de implementación 399
10.6 Métodos y clases `final` 407
10.7 Ejemplo práctico: Sistema de nómina utilizando polimorfismo 407
10.8 Ejemplo práctico: Creación y uso de interfaces 417
10.9 Clases anidadas 421
10.10 Clases de tipo de envoltura para los tipos primitivos 433
10.11 (Ejemplo práctico opcional) Acerca de los objetos: Cómo incorporar la herencia a la simulación del elevador 434
10.12 (Opcional) Descubrimiento de patrones de diseño: Introducción a los patrones de diseño de creación, estructura y comportamiento 440
10.12.1 Patrones de diseño de creación 441
10.12.2 Patrones de diseño de estructura 443
10.12.3 Patrones de diseño de comportamiento 444
10.12.4 Conclusión 445
10.12.5 Recursos en Internet y World Wide Web 446

11 Cadenas y caracteres 451
11.1 Introducción 452
11.2 Fundamentos de los caracteres y las cadenas 452
11.3 La clase `String` 453
11.3.1 Constructores de `String` 453
11.3.2 Métodos `length`, `charArt` y `getChars` de `String` 455
11.3.3 Comparación entre cadenas 456
11.3.4 Localización de caracteres y subcadenas en las cadenas 461
11.3.5 Cómo extraer subcadenas de las cadenas 463
11.3.6 Concatenación de cadenas 463
11.3.7 Métodos varios de `String` 464
11.3.8 El método `valueOf` de `String` 466
11.4 La clase `StringBuffer` 467
11.4.1 Constructores de `StringBuffer` 468
11.4.2 Los métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuffer` 468
11.4.3 Los métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuffer` 470
11.4.4 Los métodos `append` de `StringBuffer` 470
11.4.5 Los métodos `insert` y `delete` de `StringBuffer` 473
11.5 La clase `Character` 474
11.6 La clase `StringTokenizer` 479
11.7 Simulación para barajar y repartir cartas 482
11.8 Expresiones regulares, la clase `Pattern` y la clase `Matcher` 486
11.9 (Ejemplo práctico opcional) Acerca de los objetos: Manejo de eventos 495

12 Gráficos y Java2D 509
12.1 Introducción 510
12.2 Contextos de gráficos y objetos de gráficos 511
12.3 Control de colores 513
12.4 Control de tipos de letra 519
12.5 Dibujo de líneas, rectángulos y óvalos 524
12.6 Dibujo de arcos 527
12.7 Dibujo de polígonos y polilíneas 527
12.8 La API Java2D 532
12.9 (Ejemplo práctico opcional) Acerca de los objetos: Diseño de interfaces con UML 538

13 Componentes de la interfaz gráfica de usuario: Parte 1 547
13.1 Introducción 548
13.2 Generalidades de los componentes de Swing 549
13.3 `JLabel` 551
13.4 Manejo de eventos 554
13.5 Campos de texto 556
13.6 Cómo funciona el manejo de eventos 559
13.7 `JButton` 561
13.8 `JCheckBox` y `JRadioButton` 563
13.9 `JComboBox` 568
13.10 `JList` 571
13.11 Listas de selección múltiple 573
13.12 Manejo de eventos de ratón 575
13.13 Clases adaptadoras 579
13.14 Manejo de eventos de teclas 584
13.15 Administradores de esquemas 587
13.15.1 `FlowLayout` 587
13.15.2 `BorderLayout` 590
13.15.3 `GridLayout` 593
13.16 Paneles 595
13.17 (Ejemplo práctico opcional) Acerca de los objetos: Casos de uso 596

14 Componentes de la interfaz gráfica de usuario: Parte 2 607
14.1 Introducción 608
14.2 `JTextArea` 609
14.3 Creación de una subclase personalizada de `JPanel` 612
14.4 Subclase de `JPanel` que maneja sus propios eventos 615
14.5 `JSlider` 620
14.6 Ventanas: Observaciones adicionales 624
14.7 Uso de menús con marcos 625
14.8 `JPopupMenu` 632
14.9 Apariencia visual adaptable 635
14.10 `JDesktopPane` y `JInternalFrame` 638
14.11 `JTabbedPane` 642
14.12 Administradores de esquemas: `BoxLayout` y `GridBagLayout` 644
14.13 (Ejemplo práctico opcional) Acerca de los objetos: Modelo-vista-controlador 655
14.14 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.awt` y `javax.swing` 663
14.14.1 Patrones de diseño de creación 663
14.14.2 Patrones de diseño de estructura 664

http://librosysolucionarios.net

14.14.3 Patrones de diseño de comportamiento 666
14.14.4 Conclusión 669

15 Manejo de excepciones 675
15.1 Introducción 676
15.2 Generalidades acerca del manejo de excepciones 676
15.3 Ejemplo de manejo de excepciones: División entre cero 680
15.4 Jerarquía de excepciones en Java 683
15.5 Volver a lanzar una excepción 685
15.6 Cláusula `finally` 685
15.7 Limpieza de la pila 688
15.8 `printStackTrace`, `getStackTrace` y `getMessage` 690
15.9 Excepciones encadenadas 692
15.10 Declaración de nuevos tipos de excepciones 694
15.11 Constructores y manejo de excepciones 695

16 Subprocesamiento múltiple 699
16.1 Introducción 700
16.2 Estados de los subprocesos: Ciclo de vida de un subproceso 702
16.3 Prioridades y programación de subprocesos 703
16.4 Creación y ejecución de subprocesos 705
16.5 Sincronización de subprocesos 708
16.6 Relación productor/consumidor sin sincronización 709
16.7 Relación productor/consumidor con sincronización 715
16.8 Relación productor/consumidor: Búfer circular 722
16.9 Subprocesos tipo "daemon" 732
16.10 Interfaz `Runnable` 733
16.11 (Ejemplo práctico opcional) Acerca de los objetos: Subprocesamiento múltiple 738
16.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño de concurrencia 742

17 Archivos y flujos 749
17.1 Introducción 750
17.2 Jerarquía de datos 751
17.3 Archivos y flujos 752
17.4 La clase `File` 755
17.5 Creación de un archivo de acceso secuencial 760
17.6 Cómo leer datos desde un archivo de acceso secuencial 776
17.7 Actualización de archivos de acceso secuencial 782
17.8 Archivos de acceso aleatorio 782
17.9 Creación de un archivo de acceso aleatorio 783
17.10 Cómo escribir datos en forma aleatoria, en un archivo de acceso aleatorio 788
17.11 Cómo leer datos en forma secuencial, de un archivo de acceso aleatorio 793
17.12 Ejemplo práctico: Un programa para procesar transacciones 797
17.13 Nuevas APIs de E/S para la plataforma Java 809

18 Redes 825
18.1 Introducción 826
18.2 Manipulación de URLs 827
18.3 Cómo leer un archivo en un servidor Web 832
18.4 Cómo establecer un servidor simple utilizando sockets de flujo 835
18.5 Cómo establecer un cliente simple utilizando sockets de flujo 837

18.6 Interacción entre cliente/servidor mediante conexiones de socket de flujo 837
18.7 Interacción entre cliente/servidor sin conexión mediante datagramas 849
18.8 Juego de tres en raya (gato) tipo cliente/servidor, utilizando un servidor con subprocesamiento múltiple 856
18.9 La seguridad y la red 868
18.10 Servidor y cliente para conversaciones DeitelMessenger 870
18.10.1 `ServidorDeitelMessenger` y las clases de soporte 870
18.10.2 Cliente `DeitelMessenger` y las clases de soporte 878
18.11 Generalidades acerca del trabajo en red con NIO 897
18.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.io` y `java.net` 908
18.12.1 Patrones de diseño de creación 908
18.12.2 Patrones de diseño de estructura 909
18.12.3 Patrones de arquitectura 910
18.12.4 Conclusión 912

19 Multimedia: Imágenes, animación y audio 919
19.1 Introducción 920
19.2 Cómo cargar, mostrar y escalar imágenes 921
19.3 Animación de una serie de imágenes 923
19.4 Mapas de imágenes 927
19.5 Carga y reproducción de clips de audio 930
19.6 Recursos en Internet y World Wide Web 932
19.7 (Ejemplo práctico opcional) Acerca de los objetos: Animación y sonido en la vista 933

20 Estructuras de datos 951
20.1 Introducción 952
20.2 Clases autorreferenciadas 952
20.3 Asignación dinámica de memoria 953
20.4 Listas enlazadas 954
20.5 Pilas 963
20.6 Colas 967
20.7 Árboles 970

21 Paquete de utilerías de Java y manipulación de bits 997
21.1 Introducción 998
21.2 La clase `Vector` y la interfaz `Enumeration` 998
21.3 La clase `Stack` del paquete `java.util` 1002
21.4 La clase `Hashtable` 1005
21.5 La Clase `Properties` 1009
21.6 Manipulación de bits y operadores a nivel de bits 1015
21.7 La clase `BitSet` 1028

22 Colecciones 1037
22.1 Introducción 1038
22.2 Generalidades acerca de las colecciones 1039
22.3 La clase `Arrays` 1039
22.4 La interfaz `Collection` y la clase `Collections` 1043
22.5 Objetos `List` 1044
22.6 Algoritmos 1050
22.6.1 El algoritmo `sort` 1050

http://librosysolucionarios.net

22.6.2 El algoritmo `shuffle` 1054
22.6.3 Los algoritmos `reverse`, `fill`, `copy`, `max` y `min` 1056
22.6.4 El algoritmo `binarySearch` 1058
22.7 Conjuntos 1060
22.8 Mapas 1063
22.9 Envolturas de sincronización 1066
22.10 Envolturas no modificables 1066
22.11 Implementaciones abstractas 1067
22.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en el paquete `java.util` 1067
22.12.1 Patrones de diseño de creación 1067
22.12.2 Patrones de diseño de comportamiento 1067
22.12.3 Conclusión 1068

23 Conectividad de bases de datos en Java con JDBC™ 1073
23.1 Introducción 1074
23.2 Bases de datos relacionales 1075
23.3 Generalidades acerca de las bases de datos relacionales: La base de datos `libros` 1076
23.4 SQL 1079
23.4.1 Consulta básica `SELECT` 1080
23.4.2 La cláusula `WHERE` 1081
23.4.3 La cláusula `ORDER BY` 1082
23.4.4 Cómo fusionar datos de varias tablas: `INNER JOIN` 1084
23.4.5 La instrucción `INSERT` 1085
23.4.6 La instrucción `UPDATE` 1086
23.4.7 La instrucción `DELETE` 1086
23.5 Creación de la base de datos `libros` en Cloudscape 1087
23.6 Manipulación de bases de datos con JDBC 1088
23.6.1 Cómo conectarse y realizar consultas en una base de datos 1088
23.6.2 Consultas en la base de datos `libros` 1093
23.7 Procedimientos almacenados 1104
23.8 Recursos en Internet y World Wide Web 1104

24 Servlets 1111
24.1 Introducción 1112
24.2 Generalidades y arquitectura de los servlets 1114
24.2.1 La interfaz `Servlet` y el ciclo de vida de un servlet 1115
24.2.2 La clase `HttpServlet` 1116
24.2.3 La interfaz `HttpServletRequest` 1117
24.2.4 La interfaz `HttpServletResponse` 1118
24.3 Manejo de las peticiones `get` de HTTP 1118
24.3.1 Configuración del servidor Apache Tomcat 1122
24.3.2 Despliegue de una aplicación Web 1124
24.4 Manejo de peticiones `get` de HTTP que contienen datos 1127
24.5 Manejo de las peticiones `post` de HTTP 1130
24.6 Redirección de peticiones a otros recursos 1133
24.7 Aplicaciones multinivel: Uso de JDBC desde un servlet 1136
24.8 Recursos en Internet y World Wide Web 1143

25 JavaServer Pages (JSP) 1149
25.1 Introducción 1150
25.2 Generalidades acerca de las JavaServer Pages 1150

25.3 Nuestro primer ejemplo de JavaServer Page 1151
25.4 Objetos implícitos 1153
25.5 Secuencias de comandos 1154
25.5.1 Componentes de las secuencias de comandos 1155
25.5.2 Ejemplo de secuencia de comandos 1155
25.6 Acciones estándar 1158
25.6.1 La acción `<jsp:include>` 1159
25.6.2 La acción `<jsp:forward>` 1164
25.6.3 La acción `<jsp:useBean>` 1167
25.7 Directivas 1170
25.7.1 La directiva `page` 1171
25.7.2 La directiva `include` 1173
25.8 Ejemplo práctico: Libro de visitantes 1175
25.9 Recursos en Internet y World Wide Web 1185

Apéndices

A Tabla de precedencia de los operadores 1191

B Conjunto de caracteres ASCII 1193

Disponible en el CD-ROM:

C Sistemas numéricos 1195
C.1 Introducción 1196
C.2 Abreviatura de los números binarios como números octales y hexadecimales 1199
C.3 Conversión de números octales y hexadecimales a binarios 1200
C.4 Conversión de un número binario, octal o hexadecimal a decimal 1200
C.5 Conversión de un número decimal a binario, octal o hexadecimal 1200
C.6 Números binarios negativos: Notación de complemento a dos 1202

D Eventos e interfaces de escucha del elevador 1207
D.1 Introducción 1207
D.2 Eventos 1207
D.3 Componentes de escucha 1211
D.4 Repaso de los artefactos 1213

E Modelo del elevador 1215
E.1 Introducción 1215
E.2 La clase `SimulacionElevador` 1215
E.3 Las clases `Ubicacion` y `Piso` 1222
E.4 Las clases `Puerta` y `PuertaElevador` 1225
E.5 La clase `Boton` 1230
E.6 La clase `ConductoElevador` 1231
E.7 Las clases `Luz` y `Timbre` 1238
E.8 La clase `Elevador` 1241
E.9 La clase `Persona` 1249
E.10 Repaso de los artefactos 1256
E.11 Conclusión 1257

http://librosysolucionarios.net

F La vista del Elevador (en CD) 1259
F.1 Introducción 1259
F.2 Los objetos de la clase 1275
F.3 Las constantes de la clase 1277
F.4 El constructor de la clase 1278
F.5 Manejo de eventos 1280
F.5.1 Los tipos de eventos `EventoMoverElevador` 1280
F.5.2 Los tipos de eventos `EventoMoverPersona` 1280
F.5.3 Los tipos de eventos `EventoPuerta` 1281
F.5.4 Los tipos de eventos `EventoBoton` 1281
F.5.5 Los tipos de eventos `EventoTimbre` 1282
F.5.6 Los tipos de eventos `EventoLuz` 1282
F.6 Repaso de los artefactos 1282
F.7 Conclusión 1283

G Unicode® 1285
G.1 Introducción 1286
G.2 Formatos de transformación de Unicode 1287
G.3 Caracteres y glifos 1287
G.4 Ventajas/Desventajas de Unicode 1288
G.5 Sitio Web del consorcio Unicode 1289
G.6 Uso de Unicode 1290
G.7 Rangos de caracteres 1291

Bibliografía 1295

Índice 1299

Prefacio

¡Bienvenido a Java! En Deitel & Associates escribimos libros de texto sobre lenguajes de programación, tanto de nivel universitario como profesional, y nos esforzamos por mantener nuestros libros actualizados. Escribir esta quinta edición fue un placer. Este libro y el material de apoyo incluyen todo lo que necesitan los profesores y estudiantes para obtener una experiencia educativa interesante, informativa, retadora y entretenida con Java. Esta publicación concuerda con la última versión de Java [*plataforma Java 2, Standard Edition (J2SE), versión 1.4.1*] y con el diseño orientado a objetos, utilizando la versión más reciente de *UML (Lenguaje Unificado de Modelado)* desarrollado por el Object Management Group (OMG). Pusimos a tono la escritura, la pedagogía, nuestro estilo de codificación, el paquete de accesorios del libro, y además incluimos un análisis importante del desarrollo de aplicaciones de bases de datos basadas en Internet y Web. Pasamos el *Paseo por el libro* al prefacio. Este paseo ayudará a los profesores, estudiantes y profesionales a tener una idea más clara de la extensa cobertura que proporciona este libro, en relación con la programación orientada a objetos en Java, el diseño orientado a objetos con UML y el desarrollo de aplicaciones basadas en Internet y Web.

Ya sea usted profesor, estudiante, profesional experimentado o programador novato, este libro tiene mucho que ofrecerle. Java es un lenguaje de programación de clase mundial que se utiliza para desarrollar aplicaciones computacionales de uso industrial, para dispositivos que van desde teléfonos celulares y PDAs, hasta los servidores empresariales más grandes. Nosotros auditamos cuidadosamente el manuscrito, comparándolo con la *Especificación del lenguaje Java*[1], que define a este lenguaje. Como resultado, los programas que usted cree al estudiar este texto deberán funcionar con cualquier plataforma Java que sea compatible con la J2SE 1.4.1.

En este prefacio veremos la extensa suite de materiales educativos de *Cómo programar en Java, 5ª edición*, los cuales ayudarán a los profesores a maximizar la experiencia de aprendizaje de sus estudiantes en cuanto a Java. Explicamos las convenciones que utilizamos, como la presentación de la sintaxis del código de los ejemplos, para "lavar código" y para resaltar segmentos importantes de éste, lo que ayuda a los estudiantes a enfocar su atención en los conceptos clave que se introducen en cada capítulo. Daremos un vistazo general a las nuevas características de *Cómo programar en Java, 5ª edición*, incluyendo nuestra adaptación mejorada sobre la programación orientada a objetos, el desarrollo de aplicaciones Web con servlets y JSP, el ejemplo práctico opcional mejorado de diseño orientado a objetos (DOO) con UML sobre la simulación de un elevador, el vistazo general a los patrones de diseño y el uso extensivo de diagramas de UML que se han actualizado a los estándares de la versión 1.4 de UML.

El libro incluye un CD con el texto del *Kit de desarrollo de software J2SE 1.4.1* (J2SDK) de Sun Microsystems, junto con *Sun ONE Studio 4 (Community Edition)*, el entorno integrado de desarrollo (IDE) de Sun. Para un mayor apoyo a los programadores novatos, ofrecemos varias publicaciones gratuitas de la *Serie DIVE-INTO™*, las cuales explican cómo compilar, ejecutar y depurar programas en Java utilizando el Kit J2SDK, Sun

1. Existen copias electrónicas en formato HTML y PDF de la *Especificación del lenguaje Java*, las cuales están disponibles en forma gratuita en el sitio Web sobre Java de Sun Microsystems en `java.sun.com/docs/books/jls/index.html`.

http://librosysolucionarios.net

ONE Studio (Community Edition) y el software *JBuilder* de Borland, *Personal edition*. Todo este material se encuentra en inglés y está disponible en la página `www.deitel.com/books/downloads.html` o en `www.pearsoneducacion.net/deitel`, junto con otros recursos para este libro.

Además, en este sitio ofrecemos un panorama general del paquete completo de materiales adicionales para profesores y estudiantes que utilicen este libro como texto en un curso. Entre otros materiales, también en inglés, se incluye un *CD* llamado *Instructor's Resource*, con soluciones a la mayoría de los ejercicios planteados en los capítulos del libro, así como un archivo llamado *Test-Item* que contiene cientos de preguntas de opción múltiple y sus respuestas correspondientes. Los profesores que deseen mantener sesiones cerradas de laboratorio (o asignar tareas altamente estructuradas), pueden adquirir un manual de laboratorio, *Java in the Lab*. Esta publicación incluye actividades previas al laboratorio, ejercicios de laboratorio y actividades posteriores, que han sido cuidadosamente elaboradas para este fin.

Este prefacio también analiza la quinta edición de *The Java 2 Multimedia Cyber Classroom 5ª ed.*, que es la versión multimedia, en inglés, de la versión original de este libro. Esta herramienta de aprendizaje proporciona extensas características de interactividad, incluyendo hipervínculos, búsqueda de texto, "recorridos" con audio a través de los programas, animaciones en Flash® y cientos de ejercicios y sus soluciones.

Analizaremos también varias herramientas DEITEL™ de aprendizaje en línea, incluyendo una explicación del contenido de *Blackboard*, *CourseCompass* y *WebCT* de la serie *Course Management Systems*, cada uno de los cuales apoya el material de este libro.

La cuarta edición de este libro fue revisada por 35 distinguidos académicos y profesionales del ramo. Después de aplicar sus comentarios, el manuscrito para *Cómo programar en Java, 5ª edición*, fue revisado nuevamente por 44 distinguidos académicos y profesionales. En los reconocimientos mencionamos los nombres de los revisores y en dónde laboran. El prefacio concluye con información acerca de los autores y de Deitel & Associates, Inc. Si al leer este libro surge alguna pregunta, por favor envíe un correo electrónico a `deitel@deitel.com`; le responderemos inmediatamente. Visite con regularidad nuestro sitio Web, `www.deitel.com`, e inscríbase en el boletín de correo electrónico *DEITEL™ BUZZ ONLINE* en `www.deitel.com/newsletter/subscribe.html`. Utilizamos el sitio Web y el boletín para mantener a nuestros lectores actualizados en cuanto a las publicaciones y servicios de DEITEL™.

Nuevas características en Cómo programar en Java, 5ª edición

Esta edición contiene muchas nuevas características y mejoras, incluyendo:

Resaltado de código

En este libro hemos agregado mucho código resaltado (con pantalla). Al revisar nuestro código eliminamos la mayoría de los fragmentos "redundantes" que aparecían en ediciones anteriores. Dichos fragmentos los mantuvimos en la primera parte del libro como una herramienta pedagógica que ayudara a los principiantes. Queremos que el lector vea, en contexto, las nuevas características del código, por lo cual, a partir del capítulo 4, nuestras revisiones al código simplemente harán referencia a los números de línea de los nuevos segmentos de código en los programas fuente. Para facilitar a los lectores el detectar los segmentos presentados, los hemos resaltado. Esto ayuda a los estudiantes a revisar el material rápidamente cuando se preparen para algún examen.

"Lavado de código"

Lavado de código es el término que utilizamos para aplicar comentarios extensos, para utilizar identificadores importantes, para aplicar sangría y usar espaciado vertical para separar las unidades importantes de un programa. Este proceso da como resultado programas que son mucho más fáciles de leer y que sirven como autodocumentación. Hemos realizado un amplio "lavado" a todo el código de los programas de fuente de este texto, del manual de laboratorio, del material auxiliar y del *Cyber Classroom*.

El proceso para poner a punto la programación orientada a objetos en los capítulos 9 y 10

Ésta es una de las mejoras más importantes de esta nueva edición. Hemos realizado una actualización muy precisa del capítulo 9 de *Cómo programar en Java, 4ª edición*, y lo dividimos en dos capítulos. Las mejoras hacen el material más claro y accesible para los estudiantes y profesionales, especialmente para los que estudian la programación orientada a objetos por primera vez.

http://librosysolucionarios.net

Capítulo 9 "Programación orientada a objetos: herencia". El capítulo 9 lleva cuidadosamente al lector a través de una secuencia de cinco ejemplos que muestran los datos privados (`private`), protegidos (`protected`) y la reutilización de software por medio de la herencia. Comenzamos mostrando una clase con variables de instancia privada y métodos públicos para manipular esos datos. Luego implementamos una segunda clase con varias capacidades adicionales. Para esto, duplicamos gran parte del código del primer ejemplo. En nuestro tercer ejemplo empezamos nuestra explicación sobre la herencia y la reutilización de software; utilizamos la clase del primer ejemplo como una superclase y heredamos sus datos y su funcionalidad en una nueva subclase. Este ejemplo introduce el mecanismo de la herencia y demuestra que una subclase no puede acceder a los miembros privados de su superclase directamente. Esto motiva nuestro cuarto ejemplo, en el cual introdujimos datos protegidos en la superclase y demostramos que la subclase puede acceder a los datos protegidos que hereda de la superclase. El último ejemplo en la secuencia demuestra la ingeniería de software adecuada al definir los datos de la superclase como privados y al utilizar los métodos públicos de la superclase (que fueron heredados por la subclase) para manipular, desde la subclase, los datos privados de la superclase. Seguimos la introducción de cinco partes con una jerarquía de clases de tres niveles, la cual emplea las técnicas de ingeniería de software que se introducen en la primera parte de este capítulo. El capítulo cierra con una discusión de la ingeniería de software con la herencia.

Capítulo 10 "Programación orientada a objetos: polimorfismo". El nuevo capítulo 10 se basa en los conceptos de herencia que presentamos en el capítulo 9, y se enfoca en las relaciones existentes entre las clases de una jerarquía de clases. El capítulo 10 utiliza una secuencia de tres ejemplos para presentar las poderosas capacidades de procesamiento que se permiten en este tipo de relaciones. Comenzamos con un ejemplo que ilustra la relación "es un" entre un objeto de una subclase y su tipo de superclase. Esta relación permite que el objeto de la subclase sea tratado como un objeto de su superclase. Mostramos que podemos asignar una referencia de un objeto de la subclase a una variable de la superclase, e invocar los métodos de la superclase en ese objeto. Este ejemplo utiliza el polimorfismo, que permite a un programa procesar objetos de clases relacionadas por una jerarquía de clases, como objetos de su tipo de superclase. Cuando un método se invoca por medio de una variable de la superclase, se invoca la versión de ese método que es específica de la subclase. En nuestro segundo ejemplo demostramos que lo inverso no se aplica (el objeto de una superclase no se considera como un objeto de su tipo de subclase) y también mostramos que se producen errores de compilación si un programa intenta manipular un objeto de la superclase en esta forma. Nuestro tercer ejemplo demuestra que los únicos métodos que pueden invocarse a través de la variable de una superclase son los que están definidos por el tipo de superclase. El ejemplo muestra que los intentos de invocar métodos que sólo pertenecen a la subclase ocasionan errores de compilación. El capítulo continúa con un ejemplo práctico sobre el polimorfismo, en el cual procesamos un arreglo de variables que contienen referencias a objetos. Todos los objetos referenciados por los elementos del arreglo tienen una superclase abstracta común, la cual contiene el conjunto de métodos comunes para todas las clases de la jerarquía. Concluimos con un ejemplo práctico que demuestra cómo un programa que procesa objetos en forma polimórfica puede aún realizar un procesamiento específico a cada tipo, determinando el tipo del objeto que se esté procesando en ese momento.

Nuevas APIs de E/S (NIO) de Java

Las nuevas APIs de E/S de Java son un agregado considerable a la J2SE 1.4. Analizamos partes de estas APIs en algunas secciones de tres capítulos. La sección 11.8 demuestra las capacidades de las nuevas APIs de E/S en cuanto al uso de expresiones regulares, lo cual permite a los programas buscar patrones de caracteres en las cadenas. La sección 17.13 introduce las clases de E/S de alto rendimiento de las nuevas APIs de E/S, las cuales permiten a los desarrolladores aprovechar los búferes, canales, conjuntos de caracteres y demás. Esta sección también presenta un ejemplo del uso de canales y búferes para escribir datos hacia, y leer datos desde, un archivo. La sección 18.11 continúa nuestra discusión sobre las nuevas APIs de E/S con una introducción a los selectores y a la E/S sin bloqueo para implementar servidores de red de alto rendimiento. Después implementamos un programa de conversación distribuida que demuestra esas capacidades. Las secciones 11.8 y 17.13 también proporcionan vínculos Web para un estudio más detallado de las nuevas APIs de E/S.

Desarrollo de aplicaciones Web y de bases de datos con JDBC, servlets y JSP

Debido a la demanda popular, hemos regresado varios temas a *Cómo programar en Java, 5ª edición*. El capítulo 23, Conectividad de bases de datos en Java mediante JDBC, demuestra cómo crear aplicaciones controladas por

bases de datos con la API JDBC™. El capítulo 24, Servlets y el capítulo 25, JavaServer Pages™ (JSP), expanden nuestro análisis de los temas de programación relacionados con Internet y Web, y tienen todo lo que los lectores necesitan para empezar a desarrollar sus propias aplicaciones basadas en Web ¡que funcionarán en Internet! Los lectores aprenderán a crear las denominadas aplicaciones de *n* niveles, en las cuales la funcionalidad que proporciona cada nivel puede distribuirse para separar computadoras a través de Internet, o pueden ejecutarse en la misma computadora. En especial, crearemos una aplicación de encuesta de tres niveles basada en Web, y una aplicación de libro de visitantes de tres niveles basada en Web. La información de cada aplicación se almacena en el nivel de datos de la aplicación; en este libro se utiliza una base de datos implementada con el producto de bases de datos Cloudscape de IBM, basado en Java (en el CD del libro se incluye una versión de prueba). El usuario introduce las peticiones y recibe las respuestas en el nivel de cliente de cada aplicación, que por lo general es una computadora que ejecuta un navegador Web, como Microsoft Internet Explorer o Netscape. Por supuesto, los navegadores Web saben cómo comunicarse con los sitios Web a través de Internet. El nivel medio contiene un servidor Web y uno o más servlets específicos a la aplicación (en el caso de nuestra aplicación de encuesta) o JavaServer Pages (en el caso de nuestra aplicación de libro de visitantes). Utilizamos el servidor Web Tomcat de Apache como nuestro servidor de aplicaciones para estos ejemplos. Tomcat, que es la implementación de referencia para las tecnologías de servlets y JavaServer Pages, se incluye en el CD que acompaña a este libro y está disponible para descargarse en forma gratuita de `www.apache.org`. Tomcat se comunica con el nivel de cliente a través de Internet, utilizando el Protocolo de transferencia de hipertexto (HTTP). Hablamos sobre el papel imprescindible del servidor Web en la programación Web y proporcionamos muchos ejemplos que demuestran las interacciones entre un navegador Web y un servidor Web.

Lenguaje Unificado de Modelado™ (UML)

El Lenguaje Unificado de Modelado™ (UML) se ha convertido en el lenguaje de modelado gráfico preferido para diseñar sistemas orientados a objetos. En la edición anterior de este libro solamente utilizamos UML en secciones opcionales, y empleamos segmentos de diagramas de flujo convencionales y diagramas de herencia para reforzar las explicaciones. Ahora hemos transformado completamente los diagramas del libro para que sean compatibles con UML 1.4. En especial, actualizamos todas las figuras del ejemplo práctico de simulación del elevador con UML/DOO; convertimos todos los diagramas de flujo en los capítulos 4 y 5 sobre las instrucciones de control, a diagramas de actividad de UML; y convertimos todos los diagramas de herencia de los capítulos 9, 10, 12, 13 y 15 a diagramas de clases de UML.

Esta quinta edición pone a punto cuidadosamente el ejemplo práctico opcional (pero ampliamente recomendado) que presentamos sobre el diseño orientado a objetos, utilizando UML. El ejemplo práctico se envió a un equipo distinguido de revisores de DOO/UML, incluyendo líderes en el campo por parte de Rational (los creadores de UML) y del Grupo de administración de objetos (responsable del mantenimiento y la evolución de UML). En el ejemplo práctico implementamos completamente la simulación de un elevador. En las secciones "Acerca de los objetos" al final de los capítulos 1 a 8, 10 a 14, 16 y 19, presentamos una introducción detallada al diseño orientado a objetos, utilizando UML. Mostramos un subconjunto conciso y simplificado de UML, y después guiamos al lector a través de su primera experiencia con el diseño, la cual está dirigida a diseñadores y programadores orientados a objetos que apenas inician. El ejemplo práctico está completamente resuelto. No es un ejercicio, sino una experiencia completamente educativa, que concluye con un recorrido detallado por el código de Java. En cada uno de los primeros cinco capítulos nos concentramos en la metodología "convencional" de la programación estructurada, ya que los objetos que creamos utilizan estas piezas de programa estructurado. Concluimos cada capítulo con una sección "Acerca de los objetos", en la que presentamos una introducción al diseño orientado a objetos (DOO) mediante el uso de UML. Estas secciones ayudarán a los estudiantes a desarrollar un diseño orientado a objetos, de manera que puedan utilizar inmediatamente los conceptos de programación orientada a objetos que comienzan a aprender en el capítulo 8. En la primera de estas secciones al final del capítulo 1, presentamos los conceptos básicos y la terminología del DOO. En las secciones "Acerca de los objetos" opcionales al final de los capítulos 2 a 5 consideramos cuestiones más importantes mientras hacemos frente a problemas que suponen un reto con las técnicas del DOO. Analizamos la declaración de un problema típico en el que se requiere la construcción de un sistema, determinamos los objetos necesarios para implementarlo, los atributos que necesitan tener estos objetos, y los comportamientos que necesitan exhibir dichos objetos y especificamos la manera en que los objetos deberán interactuar entre sí para cumplir con los requerimientos del sistema. Logramos esto incluso antes de comenzar a discutir cómo escribir programas en Java. En

los apéndices D a F, incluimos una implementación en Java del sistema orientado a objetos que diseñamos en los capítulos anteriores. Este ejemplo práctico ayudará a los estudiantes a prepararse para los tipos de proyectos que encontrarán en la industria. Empleamos un proceso de diseño orientado a objetos incremental y cuidadosamente desarrollado para producir un modelo en UML para nuestra simulación de elevador. A partir de este diseño producimos una verdadera y funcional implementación en Java utilizando las nociones claves de programación, incluyendo clases, objetos, encapsulamiento, visibilidad, composición y herencia.

Descubrimiento de patrones de diseño

Estas secciones opcionales introducen patrones de diseño orientado a objetos populares. Durante la década pasada, la industria de la ingeniería de software progresó considerablemente en el campo de los *patrones de diseño*: arquitecturas comprobadas para construir software orientado a objetos que sea flexible y pueda mantenerse[2]. El uso de patrones de diseño puede reducir considerablemente la complejidad del proceso de diseño. Presentamos varios patrones de diseño en Java, pero éstos pueden implementarse en cualquier lenguaje orientado a objetos tal como C++, C# o Visual Basic .NET. Describimos varios patrones de diseño utilizados por Sun Microsystems en la API de Java. Además, se utilizaron en muchos programas de este libro, los cuales identificaremos en las secciones "Descubrimiento de patrones de diseño". Estos programas proporcionan ejemplos acerca del uso de patrones de diseño para construir software orientado a objetos que sea confiable y robusto.

Método de enseñanza

Esta quinta edición contiene una vasta colección de ejemplos, ejercicios y proyectos sustraídos de diversos campos para proporcionar al estudiante la oportunidad de resolver problemas interesantes del mundo real. El libro se concentra en los principios de la buena ingeniería de software, haciendo hincapié en la claridad de los programas. Evitamos la tecnología arcana y las especificaciones sintácticas, para favorecer la enseñanza mediante ejemplos. Nuestros ejemplos de código se han probado en las plataformas populares de Java. Somos educadores que enseñamos temas de vanguardia en salones de clases de la industria alrededor del mundo. Este texto pone énfasis en la buena pedagogía.

Cómo aprender Java mediante el método LIVE-CODE™

Este libro contiene numerosos ejemplos "reales". Cada nuevo concepto se presenta en el contexto de un ejemplo completo y funcional, que es seguido inmediatamente por una o más ejecuciones que muestran la entrada/salida del programa. Este estilo ejemplifica la manera en que enseñamos y escribimos acerca de la programación, y es el enfoque de nuestros cursos multimedia *Cyber Classrooms* y de capacitación basados en Web. A este método de enseñar y escribir le llamamos **Método del código activo** (o ***Método LIVE-CODE™***). *Utilizamos lenguajes de programación para enseñar lenguajes de programación.* Leer los ejemplos en el texto es muy parecido a escribirlos y ejecutarlos en la computadora. Tanto en el CD que acompaña al libro como en `www.deitel.com` encontrará todo el código fuente para los ejemplos que aparecen en el libro. Le recomendamos ejecutar todos los ejemplos.

¡Programación en Java con aplicaciones y Swing desde el capítulo dos!

Este libro lo pasa directamente a la programación de aplicaciones Java con los componentes de la GUI Swing del capítulo 2. Hay mucho que hacer en Java, así que ¡manos a la obra! Definitivamente Java no es algo trivial, pero es divertido programar en este lenguaje y los estudiantes pueden ver resultados inmediatos. Además, pueden ejecutar rápidamente programas gráficos, animados, basados en multimedia, con uso intensivo de audio, con subprocesamiento múltiple, con uso intensivo de bases de datos, o programas basados en red, a través de las extensas bibliotecas de clases de Java que cuentan con componentes reutilizables. Los alumnos pueden implementar proyectos impresionantes. Por lo general son más creativos y productivos en un curso de uno o dos semestres que en cursos de introducción a C y C++.

2. Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software.* (Massachusetts: Addison-Wesley, 1995.)

http://librosysolucionarios.net

Acceso a World Wide Web

Todos los ejemplos de código fuente (en inglés) para *esta quinta edición* (y demás publicaciones de los autores) están disponibles en Internet, de manera que pueden descargarse del siguiente sitio Web:

```
www.deitel.com
www.pearsoneducacion.net/deitel
```

Puede registrarse en forma rápida y sencilla, y las descargas son gratuitas. Hacer cambios a los ejemplos e inmediatamente ver los efectos de esos cambios es una excelente manera de mejorar su aprendizaje de Java.

Objetivos

Cada capítulo comienza con objetivos que informan a los estudiantes lo que deben esperar, y les brinda la oportunidad, después de leer el capítulo, de determinar si cumplieron los objetivos planteados. Los objetivos sirven para crear confianza.

Frases

Los objetivos de los capítulos van seguidos de una o más frases. Algunas son graciosas, otras, filosóficas y algunas más ofrecen ideas interesantes. Hemos descubierto que los estudiantes disfrutan relacionando las frases con el material del capítulo. Muchas de éstas merecen leerse otra vez, *después* de leer los capítulos.

Plan general

El plan general de cada capítulo permite a los estudiantes abordar el material de manera ordenada. Junto con los objetivos de cada capítulo, el plan general ofrece a los estudiantes un panorama de los temas a tratar para establecer un ritmo cómodo y efectivo de aprendizaje.

23,341 líneas de código en 219 programas de ejemplo (con la salida de los programas)

Presentamos las características de Java en un contexto de programas completos y funcionales en Java. Estos programas LIVE-CODE ™ varían en longitud: desde unas cuantas líneas de código hasta ejemplos importantes con cientos de líneas de código. Cada programa va seguido por una ventana que contiene los resultados que se producen al ejecutar dicho programa; de manera que los estudiantes puedan confirmar que los programas funcionen como se esperaba. El proceso de comparar las operaciones de salida con las instrucciones del programa que producen esas operaciones es una excelente manera de aprender y reforzar los conceptos. Nuestros programas están diseñados para ejercitar las diversas características de Java.

615 Ilustraciones/Figuras

Incluimos una gran cantidad de gráficas, dibujos lineales, programas y salidas de programa. Hemos convertido todos los diagramas de flujo en diagramas de actividad en UML. También utilizamos diagramas de clase en UML para modelar las relaciones existentes entre las clases a lo largo de este texto.

534 Tips de programación

Hemos incluido tips de programación para ayudar a los estudiantes a enfocarse en los aspectos importantes del desarrollo de programas. Resaltamos cientos de estos tips como *Buenas prácticas de programación*, *Errores comunes de programación*, *Tips para prevenir errores*, *Observaciones de apariencia visual*, *Tips de rendimiento*, *Tips de portabilidad* y *Observaciones de ingeniería de software*. Estos tips y prácticas representan lo mejor que hemos podido recabar a lo largo de seis décadas (combinadas) de experiencia en la programación y la enseñanza. Uno de nuestros estudiantes, especialista en matemáticas, recientemente nos comentó que siente que este método es similar al de resaltar axiomas, teoremas y corolarios en los libros de matemáticas, ya que proporciona una base sólida sobre la cual se puede construir buen software.



82 Buenas prácticas de programación

Las Buenas prácticas de programación son tips que llaman la atención hacia técnicas para escribir programas claros. Además, estas técnicas ayudan a los estudiantes a producir programas más legibles, más fáciles de entender y mantener.



156 Errores comunes de programación

Los estudiantes que están aprendiendo un lenguaje (especialmente en su primer curso de programación) tienden con frecuencia a cometer ciertos tipos de errores. Al poner atención en estos Errores comunes de programación, se reduce la probabilidad de que los estudiantes cometan los mismos errores. ¡También hace las líneas más cortas afuera de las oficinas de los profesores!



50 Tips para prevenir errores

Cuando diseñamos por primera vez este "tipo de tip", pensamos que lo usaríamos estrictamente para decir a las personas cómo probar y depurar programas en Java. De hecho, muchos de los tips describen aspectos de Java que reducen la probabilidad de "errores", lo que por consecuencia simplifica los procesos de prueba y depuración.



36 Observaciones de apariencia visual

Le ofrecemos Observaciones de apariencia visual para resaltar las convenciones de la interfaz gráfica de usuario. Estas observaciones ayudan a los estudiantes a diseñar sus propias interfaces gráficas de usuario en conformidad con las normas de la industria.



52 Tips de rendimiento

A lo largo de nuestra experiencia, enseñar a los estudiantes a escribir programas claros y comprensibles es, sin duda, la meta más importante para el primer curso de programación. Pero los estudiantes quieren escribir los programas que sean más rápidos, que usen la menor cantidad de memoria, requieran el menor número de pulsaciones de tecla, o que destaquen de alguna otra manera. Los estudiantes realmente se preocupan por el rendimiento. Quieren saber qué es lo que pueden hacer para mejorar el rendimiento de sus programas. Es por ello que nosotros resaltamos las oportunidades para mejorar el rendimiento de los programas: hacer que los programas se ejecuten más rápido o minimizar la cantidad de memoria que ocupan.



23 Tips de portabilidad

Una de las características que supuestamente han llevado a Java a la fama es su portabilidad "universal", por lo que algunos programadores asumen que si implementan una aplicación en Java, ésta será "perfectamente" portable a través de todas las plataformas Java, en forma automática. Desafortunadamente no siempre es el caso. Incluimos Tips de portabilidad para ayudar a los estudiantes a escribir código portable, y para ofrecer ideas acerca de cómo puede Java lograr su alto nivel de portabilidad. Cómo programar en Java tiene menos de estos tips que nuestros libros, Cómo programar en C y Cómo programar en C++ debido a que Java está diseñado para ser portable de arriba abajo; y para lograr esta portabilidad, se requiere de un menor esfuerzo por parte del programador de Java que del que trabaja con C o C++.



135 Observaciones de ingeniería de software

El paradigma de la programación orientada a objetos requiere de una completa reconsideración sobre la manera en que construimos los sistemas de software. Java es un lenguaje efectivo para lograr una buena ingeniería de software. Las Observaciones de ingeniería de software resaltan los asuntos de arquitectura y diseño, lo cual afecta la construcción de los sistemas de software, especialmente los de gran escala. Mucho de lo que los estudiantes aprendan aquí les será útil tanto en cursos de nivel superior como en la industria, a medida que empiecen a trabajar con sistemas grandes y complejos del mundo real.

Resumen (954 puntos de resumen)

Cada capítulo termina con elementos pedagógicos adicionales. Presentamos un resumen completo del capítulo, en forma de lista. Cada capítulo contiene un promedio de 38 puntos de resumen. Esto ayuda a los estudiantes a revisar y reforzar los conceptos clave.

Terminología (2166 términos)

En la sección *Terminología* incluimos una lista en orden alfabético de los términos importantes definidos en el capítulo; de nuevo, esto sirve como un refuerzo adicional. Cada capítulo contiene un promedio de 87 términos.

437 ejercicios de autoevaluación y sus respuestas (la cuenta incluye partes separadas)

En esta sección incluimos diversos ejercicios de autoevaluación así como sus respuestas, para que los estudiantes practiquen el autoestudio. Esto brinda la oportunidad de familiarizarse con el material y prepararse para intentar resolver los ejercicios regulares. Se debe alentar a los estudiantes para que hagan todos los ejercicios de autoevaluación y comprueben sus respuestas.

858 ejercicios (la cuenta incluye partes separadas)

Cada capítulo concluye con un conjunto de ejercicios que incluyen un sencillo recordatorio de terminología y conceptos importantes; escritura de instrucciones individuales en Java; escritura de pequeñas porciones de métodos y clases en Java; escritura de métodos, clases, aplicaciones y subprogramas completos en Java; y escritura de proyectos finales. El gran número de ejercicios, a lo largo de una extensa variedad de áreas, permite a los profesores diseñar sus cursos de acuerdo con las necesidades específicas de sus audiencias, y modificar las asig-

http://librosysolucionarios.net

naturas del curso cada semestre. Los profesores pueden usar estos ejercicios para asignar tareas, realizar exámenes cortos y los exámenes principales. Las soluciones para la mayoría de los ejercicios se incluyen en el *CD de recursos para el instructor*, el cual *está disponible sólo para profesores*, a través de los representantes de Pearson Educación. **[NOTA: No nos escriba para solicitarnos el CD del instructor. La distribución de este accesorio está limitada estrictamente a profesores universitarios que tienen este libro como texto de sus cursos. Los profesores sólo pueden obtener el manual de soluciones a través de los representantes de esta empresa.]** Los estudiantes y lectores pueden obtener la solución de aproximadamente la mitad de los ejercicios si compran la quinta edición del *Java 2 Multimedia Cyber Classroom*, 5ª edición, producto en inglés que ofrece además muchas otras opciones muy valiosas, y es ideal ya sea como autoestudio o referencia. También encontrará disponible en nuestro sitio Web (`www.deitel.com`) otros productos de apoyo a la quinta edición de este libro, los cuales están disponibles en tiendas de libros en línea como `www.informIT.com`. Si usted ya tiene el libro de texto, puede comprar el producto *Java 2 Multimedia Cyber Classroom, 5ª edición* (ISBN# 0-13-101769-1) por separado en `www.InformIT.com/cyberclassrooms`.

Bibliografía

Se incluye en este libro una extensa bibliografía de libros, artículos y documentación de Java 2 por parte de Sun Microsystems, para que el lector pueda profundizar en los temas aquí descritos.

Software incluido con este libro

Hay una variedad de herramientas de desarrollo en Java, disponibles para su venta. Aunque no son necesarias para empezar a trabajar con Java. Escribimos este libro utilizando solamente el *Kit de desarrollo de software Java 2 Standard Edition (J2SDK), versión 1.4.1*. Para su conveniencia, este kit se incluye en el CD del libro. Siempre hay una versión actualizada del J2SDK que puede descargarse del sitio Web Java de Sun en `java.sun.com/j2se`. De este sitio también se puede descargar la documentación del J2SDK.

Gracias a la cooperación de Sun, también pudimos incluir en el CD un poderoso entorno de desarrollo integrado (IDE) de Java: *SunONE™ Studio 4, Community Edition*. Éste es un IDE profesional escrito en Java, que incluye un diseñador de interfaz gráfica de usuario, editor de código, compilador, depurador visual y mucho más. Hay que instalar el J2SDK primero, antes de instalar *Sun ONE™ Studio 4 Community Edition*. Si tiene preguntas acerca del uso de este software puede leer la documentación que viene en el CD, o puede también leer la publicación, en inglés, *Dive Into Sun ONE Studio 4, Community Edition* de la serie *DIVE-INTO™*. Este documento está disponible en la página `www.deitel.com/books/downloads.html`, junto con los recursos para esta quinta edición.

El CD contiene los ejemplos del libro (incluyendo la implementación del ejemplo práctico del elevador) y una página Web HTML con vínculos a los sitios Web de Deitel & Associats, Inc.. Si usted cuenta con acceso a Internet, puede cargar esta página Web en su navegador para que obtenga un rápido acceso a todos los recursos. Además incluimos varios capítulos y apéndices de otras publicaciones Deitel. Se incluye material sobre XHTML y Hojas de estilo en cascada (para usarse con el capítulo 24, Servlets y con el capítulo 25, JavaServer Pages™). Además se incluye material sobre el Lenguaje de marcado extendible (XML) y las APIs de Java para procesar XML, que ahora forman parte del J2SE 1.4.

Accesorios para la quinta edición de Cómo programar en Java

Este libro cuenta con diversos accesorios, todo en inglés, para los profesores. El CD llamado *Instructor's Resource CD (IRCD)* contiene el *Manual del instructor* con las soluciones a la gran mayoría de los ejercicios que se encuentran al final de cada capítulo, y un archivo llamado *Test Item File* con preguntas de opción múltiple (aproximadamente dos por cada sección del libro). Además incluye diapositivas de PowerPoint con todo el código y las figuras del libro, así como una lista de los elementos que resumen los puntos clave del texto. Los profesores pueden adaptar las diapositivas según sus requerimientos. Estas diapositivas de PowerPoint pueden descargarse del sitio `www.deitel.com`, donde se ofrecen recursos tanto para profesores como para alumnos. Para los profesores, el *Sitio Web Companion* ofrece material (*Syllabus Manager*) que les ayudará a planear los cursos en forma interactiva, y a crear programas de estudios en línea.

http://librosysolucionarios.net

Los estudiantes también pueden beneficiarse de la funcionalidad del *Sitio Web Companion*. Los recursos que se incluye para este libro son:

- Diapositivas adaptables de PowerPoint®.
- Ejemplos de código fuente.
- Materiales de referencia de los apéndices del libro (como una tabla de precedencia de operadores, conjunto de caracteres y recursos Web).

Los recursos de cada capítulo del libro para los estudiantes son:

- Objetivos de cada capítulo.
- Elementos destacados (por ejemplo, resumen del capítulo).
- Plan general.
- Tips (por ejemplo, *Errores comunes de programación*, *Tips para prevenir errores*, *Buenas prácticas de programación*, *Observaciones de apariencia visual*, *Tips de portabilidad*, *Tips de rendimiento* y *Observaciones de ingeniería de software*).
- Guía de estudio en línea, la cual contiene ejercicios adicionales de autoevaluación que incluye las respuestas cortas (por ejemplo, verdadero/falso) y proporciona una retroalimentación inmediata al estudiante.

El *Sitio Web Companion* con todo el material anterior, en idioma inglés, se encuentra en `www.pearsoneducacion.net/deitel`.

Java in the Lab

Este manual de laboratorio (cuyo título completo es *Java in the Lab, Lab Manual to Accompany Java How to Program, 5ª edición*; ISBN 0-13-101631-8[3]) complementa a este libro (*Cómo programar en Java*), y al producto opcional *Java 2 Multimedia Cyber Classroom, 5ª edición*, mediante una serie de tareas de laboratorio diseñadas para reforzar la comprensión del estudiante en cuanto al material de lectura. Este manual de laboratorio está diseñado para laboratorios cerrados, que consisten en clases programadas en forma regular y supervisadas por un profesor. Los laboratorios cerrados ofrecen un excelente ambiente para el aprendizaje, ya que los estudiantes pueden utilizar los conceptos presentados en clase para resolver problemas de laboratorio cuidadosamente diseñados. Así, los profesores tienen más posibilidades de evaluar la comprensión de sus estudiantes con respecto al material, supervisando el progreso de éstos en el laboratorio. Este manual de laboratorio puede usarse también para laboratorios abiertos, tareas y autoestudio.

Java in the Lab se enfoca en los capítulos 1 a 12, 15 y 17 de este libro. Cada capítulo del manual se divide en 3 partes: *Actividades previas al laboratorio (Prelab Activities)*, *Ejercicios de laboratorio (Lab Exercises)* y *Actividades posteriores al laboratorio (Postlab Activities)*.[4] Cada capítulo contiene objetivos que introducen los temas clave del laboratorio, junto con una lista de asignaciones que permite a los estudiantes marcar cuáles ejercicios ha asignado el profesor. Cada página del manual del laboratorio está perforada, de forma que los estudiantes puedan entregar sus respuestas (en caso de ser necesario). Nuevamente, es prudente mencionar que todo el material complementario mencionado en este prefacio se encuentra en idioma inglés.

Las soluciones a los ejercicios de las secciones antes mencionadas se encuentran en formato electrónico. Los profesores pueden obtener estos materiales si los solicitan a los representantes de esta editorial; las soluciones no están disponibles para los estudiantes.

---

3. *Cómo programar en Java, 5ª edición*, y el manual de laboratorio también están disponibles en conjunto a través de un paquete especial (ISBN# 0-13-102719-0).
4. Las prácticas de los capítulos 12, 15 y 17 no contienen los extensos conjuntos de actividades disponibles en los capítulos anteriores. Sin embargo, los profesores podrán llevar a cabo las prácticas de laboratorios en forma efectiva, utilizando los ejercicios que hemos incluido sobre estos temas más complejos. Los profesores con requerimientos especiales pueden escribir a `deitel@deitel.com`.

Actividades previas al laboratorio

Se pretende que los estudiantes resuelvan las actividades de la sección *Prelab Activities* después de estudiar cada capítulo del presente libro. Estas actividades evalúan la comprensión de los estudiantes en cuanto al material presentado en el libro de texto, y los prepara para los ejercicios de programación de la sesión de laboratorio. Los ejercicios se enfocan en terminología y conceptos de programación importantes, y son efectivos para autoevaluación. Las actividades previas al laboratorio incluyen *Ejercicios de asociación*, *ejercicios de llenar espacios en blanco*, *preguntas de respuestas cortas*, *ejercicios de programación* (éstos solicitan a los estudiantes que determinen qué es lo que hacen pequeños segmentos de código, sin necesidad de ejecutar el programa) y *ejercicios de corrección de código* (éstos piden a los estudiantes que identifiquen y corrijan todos los errores en pequeños segmentos de código).

Ejercicios de laboratorio

La sección más importante de cada capítulo es la que corresponde a *Lab Exercises*. Estos ejercicios enseñan a los estudiantes cómo aplicar lo aprendido en *Cómo programar en Java, 5ª edición*, y los preparan para escribir programas en Java. Cada laboratorio contiene uno o más ejercicios y un problema de depuración. Los ejercicios de laboratorio contienen lo siguiente:

- Los *objetivos del laboratorio* (Lab Objectives) resaltan los conceptos específicos sobre los cuales se enfoca el ejercicio de laboratorio.
- Las *descripciones del problema* (Problem Descriptions) proporcionan los detalles del ejercicio y sugerencias que ayudan a los estudiantes a implementar el programa.
- Las *salidas de muestra* (Sample Outputs) ilustran el comportamiento esperado del programa, el cual favorece a la descripción de los problemas y ayuda a los estudiantes a escribir sus programas.
- Las *plantillas de programa* (Program Templates) toman los programas completos de Java y reemplazan líneas clave de código con comentarios que describen el código faltante.
- Los *tips para la solución del problema* (Problem-Solving Tips) resaltan los puntos clave que los estudiantes necesitan considerar cuando resuelven los ejercicios del laboratorio.
- Las *preguntas y actividades de seguimiento* (Follow-Up Questions and Activities) piden a los estudiantes que modifiquen las soluciones a los ejercicios, que escriban nuevos programas que sean similares a las soluciones de sus ejercicios de laboratorio o que expliquen las opciones de implementación que se eligieron cuando resolvieron los ejercicios de laboratorio.
- Los *problemas de depuración* (Debugging Problems) consisten de bloques de código que contienen errores de sintaxis y/o errores lógicos. Esto alerta a los estudiantes sobre los posibles tipos de errores que pueden encontrar durante el proceso de programación.

Actividades posteriores al laboratorio

Por lo general, los profesores asignan actividades posteriores al laboratorio para reforzar los conceptos importantes o para que los estudiantes adquieran más experiencia en programación fuera del laboratorio. La sección *Postlab Activities* evalúa el conocimiento del alumno sobre el material previo al laboratorio y los ejercicios del mismo, y pide al alumno que aplique el conocimiento adquirido para crear programas desde cero. Esta sección proporciona dos tipos de actividades de programación: ejercicios de código y retos de programación. Los ejercicios de código son cortos y sirven como repaso después de haber cumplido las *actividades previas al laboratorio* y los *ejercicios de laboratorio*. Estos ejercicios piden a los estudiantes que escriban programas o segmentos de programas que utilicen los conceptos importantes del libro. Los *retos de programación* permiten a los estudiantes aplicar el conocimiento que han adquirido en clase en ejercicios sustanciales de programación. Las sugerencias, las salidas de muestra y/o el pseudocódigo se proporcionan para ayudar a los estudiantes con estos problemas. Los estudiantes que completan de manera exitosa los *retos de programación* de un capítulo ciertamente han dominado el material del mismo. Las respuestas a los retos de programación están disponibles para su descarga en `www.deitel.com/books/downloads.html`.

http://librosysolucionarios.net

Java 2 Multimedia Cyber Classroom, 5ª edición, y The Complete Java 2 Training Course, 5ª edición

Hemos actualizado nuestra versión multimedia interactiva opcional del libro —*The Java 2 Multimedia Cyber Classroom, 5ª edición* (CD para Windows®)— con bastante audio adicional, incluyendo el nuevo material sobre desarrollo de bases de datos con JDBC y desarrollo de aplicaciones Web con servlets y JavaServer Pages. Este recurso está cargado con características electrónicas que son ideales tanto para el aprendizaje como de referencia. El número ISBN del *Cyber Classroom* es 0-13-101769-1. Los *Cyber Classrooms* de Deitel™ por lo general están disponibles en CD y en distintos tipos de formatos de capacitación en Web.

El CD que viene con este manual proporciona una introducción en la cual los autores revisan las características del *Cyber Classroom*. Los 219 programas LIVE-CODE™ de ejemplo que vienen en el libro de texto, verdaderamente "cobran vida" en el *Cyber Classroom*. Si desea modificar un programa y ver el efecto de sus cambios, al hacer clic en el icono del disco flexible el código fuente "despega" del CD y "cae dentro" de uno de sus propios directorios, de manera que pueda editarlo, recompilar el programa y ejecutar su nueva versión. Además, el *Cyber Classroom* contiene el texto completo (en inglés) de *Cómo programar en Java, 5ª edición*, en un formato especial para búsqueda de texto.

Asimismo, el *Cyber Classroom* proporciona exámenes de autoevaluación (con respuestas) para cada capítulo del libro. Estos exámenes son poderosas características que permiten a los usuarios medir su comprensión de los conceptos de programación que presentan los capítulos. Cada pregunta del examen tiene un hipervínculo a la sección, dentro del libro, de donde se deriva la pregunta. Esto permite a los usuarios revisar el material apropiado del capítulo antes o después de responder a la pregunta. Además se proporciona un cuadro que presenta los resultados, por capítulo, de cada examen del usuario.

El *Cyber Classroom* se basa en formato estilo navegador, de manera que recuerda las secciones que usted visitó recientemente y le permite moverse hacia delante o hacia atrás entre ellas. Las miles de entradas del índice están hipervinculadas al texto correspondiente. Las entradas de la tabla de contenido son vínculos dinámicos, de manera que al hacer clic en el nombre de un capítulo o sección lo llevará de inmediato al texto de referencia.

A los estudiantes les fascinará el hecho de que el *Cyber Classroom* ofrece las soluciones de cerca de la mitad de los ejercicios del libro. Estudiar y ejecutar estos programas adicionales es una excelente manera de que los estudiantes mejoren su experiencia de aprendizaje con código activo.

Para una lista completa de los *Cyber Classrooms* y *Complete Training Courses* disponibles y futuros, visite el sitio Web de los autores en: `www.deitel.com` o `www.InformIT.com/deitel`.

Advanced Java™ 2 Platform How to Program

Nuestro libro complementario, en inglés, ***Advanced Java 2 Platform How to Program*** se enfoca en la ***Plataforma Java 2, Enterprise Edition (J2EE)***, y presenta características avanzadas de esta plataforma de Java e introduce la ***Plataforma Java 2, Micro Edition (J2ME)***. Este libro está destinado a los desarrolladores y estudiantes universitarios de nivel superior en cursos avanzados, que ya conocen Java y desean profundizar más sobre este lenguaje. Además presenta nuestro libro distintivo LIVE-CODE™, con programas funcionales completos y más de 37,000 líneas de código. Los programas son más complejos que los presentados en *Cómo programar en Java, 5ª edición*. Este libro expande la cobertura de la Conectividad de bases de datos en Java (JDBC), los servlets y las JavaServer Pages (JSP) expuestas en *Cómo programar en Java, 5ª edición*. También cubre las tecnologías emergentes y más avanzadas de Java, que son de interés para los desarrolladores de aplicaciones empresariales, incluyendo el Modelo-Vista-Controlador; Java 2D y Java 3D, Modelo de componentes JavaBeans; Seguridad; Java 2 Micro Edition (J2ME) e Internet inalámbrica; Llamada a métodos remotos (RMI); Enterprise JavaBeans (EJBs); Servicio de mensajes de Java (JMS); Jini; JavaSpaces; Jiro; Extensiones de gestión de Java (JMX); Arquitectura común del agente de petición de objetos (CORBA); Redes de igual a igual; Servicios Web; XML e Interfaz nativa de Java (JNI).

Java Web Services for Experienced Programmers

Este libro que forma parte de la nueva serie *Deitel Developer* para profesionales de computación, utiliza nuestro conocido método LIVE-CODE™ para evaluar las más recientes tecnologías de XML y Java para crear e

integrar servicios Web, incluyendo el *Paquete para desarrolladores de Servicios Web en Java (Java Web Services Developer Pack)*. El libro está dirigido a los profesionales de la industria que requieren de una cobertura detallada sobre las tecnologías de servicios Web en Java. También es adecuado para los cursos de ciencias computacionales de nivel superior que tratan sobre los servicios Web, o como suplemento para los cursos en computación distribuida y programación avanzada en Java. Los temas que se cubren en este libro son: XML; Definiciones de tipos de documentos (DTDs); Modelo de objetos de documentos (DOM™) y la API de Java para procesamiento de XML (JAXP); Transformaciones de hojas de estilo extensibles (XSLT™) y las API de transformación para XML (TrAX); Protocolo simple de acceso a objetos (SOAP); Lenguaje de descripción de servicios Web (WSDL); Integramiento, descubrimiento y descripción universal (UDDI) y la API de Java para Registros de XML (JAXR); API de Java para Llamadas a procedimientos remotos basadas en XML (JAX-RPC); API de Java para Mensajería XML (JAXM) y la API SOAP con adjuntos para Java (SAAJ); Seguridad de los servicios Web con el Nivel de sockets seguro (SSL); Firma digital XML (XML Signature), Cifrado con XML (XML Encryption), Especificación de gestión de claves con XML (XKMS), Lenguaje de marcado de aserciones de seguridad (SAML) y Lenguaje de marcado de control de acceso extendible (XACML); y los Servicios Web inalámbricos con Java 2 Micro Edition (J2ME™).

La serie Course Management Systems: Blackboard™, WebCT™, CourseCompass℠ y Premium CourseCompass℠

Parte del contenido de la serie introductoria para lenguajes de programación *Cómo programar* de Deitel, incluyendo este libro, se ha integrado en varios de los populares textos de la serie Course Management Systems, como en Blackboard, CourseCompass y WebCT. Una versión mejorada de CourseCompass, llamada Premium Course Compass, estará disponible para la quinta edición de *Cómo programar en Java*. Estos textos ayudan en la creación, el manejo y el uso de sofisticadas herramientas y programas de educación basados en Web. Con ese material, creado por y para profesores, los profesores pueden ahorrarse horas de introducción de datos. Blackboard, CourseCompass and WebCT ofrecen:

- **Características para crear y personalizar cursos en línea**, como zonas para colocar información del curso (por ejemplo, políticas, planes de estudio, anuncios, asignaturas, grados, evaluaciones de rendimiento y seguimiento del proceso), herramientas de administración de clases y estudiantes, un libro de calificaciones, herramientas para reportes, rastreo de páginas, un calendario y asignaturas.
- **Herramientas de comunicaciones** para ayudar a crear y mantener las relaciones interpersonales entre los estudiantes y los profesores, incluyendo salones de conversación, pizarras electrónicas, documentación compartida, foros de discusión y correo electrónico privado.
- **Herramientas de evaluación flexibles** que permiten al profesor crear cuestionarios en línea y exámenes vinculados de manera directa al texto, y que califican y dan seguimiento a los resultados con efectividad. Todas las evaluaciones pueden introducirse en el libro de calificaciones para una eficiente administración del curso. Además, WebCT permite a los instructores administrar el tiempo de los cuestionarios en línea.
- El **Material de apoyo** para los instructores se encuentra disponible en formato para impresión y para consulta en línea.

Además de las herramientas que se encuentran en Blackboard y WebCT, CourseCompass incluye:

- **La página de inicio de CourseCompass**, la cual hace el curso tan fácil como navegar en este libro. Una tabla de contenido expandible que permite a los instructores ver el contenido del curso al instante y vincularse con cualquier sección.
- **Secciones de ayuda en línea "How Do I"**, están disponibles para los usuarios que requieren sitios con ayuda personalizada para los cursos, incluyendo las instrucciones paso a paso para agregar diapositivas de Power-Point®, vídeo y más.
- **Guía del instructor Quick Start**, ofrece a los profesores la opción de crear cursos en línea mediante un sencillo proceso, paso a paso.

Para demostraciones gratuitas en línea y aprender más acerca de la serie Course Management Systems, que cuenta con material de apoyo para este libro, visite las páginas Web:

- Blackboard: `www.blackboard.com`
- WebCT: `www.webct.com`
- CourseCompass: `www.coursecompass.com`

La nueva serie para desarrolladores (DEITEL™ Developer)

Deitel & Associates, Inc., está realizando un compromiso considerable por cubrir las tecnologías de punta para los profesionales de software de la industria, a través del lanzamiento de nuestra serie para desarrolladores *DEITEL™ Developer Series*. *Web Services A Technical Introduction* y *Java Web Services for Experienced Programmers* se encuentran entre los primeros libros en la serie. A estos libros le seguirán *Java 2 Enterprise Edition*, *Java 2 Micro Edition*, *.NET A Technical Introduction*, *ASP .NET with Visual Basic .NET for Experienced Programmers*, *ASP .NET with C# for Experienced Programmers* y muchos más. Por favor visite el sitio `www.deitel.com` para ver las actualizaciones en todos los títulos publicados y futuros de esta serie.

Paseo por el libro

Está a punto de estudiar uno de los lenguajes de programación más excitantes y con un rápido desarrollo en la actualidad. El dominio de Java le ayudará a desarrollar poderoso software de aplicaciones comercial y personal. En esta sección daremos un rápido vistazo a muchos de los temas y características de Java que veremos a lo largo de todo el libro.

Capítulo 1 "Introducción a las computadoras, Internet y Web": Habla sobre lo que son las computadoras, cómo trabajan y cómo se programan. Este capítulo presenta una breve historia sobre el desarrollo de los lenguajes de programación, partiendo desde los lenguajes máquina, pasando por los lenguajes ensambladores y terminando con los lenguajes de alto nivel. Se habla también sobre los orígenes del lenguaje de programación Java. El capítulo incluye una introducción a un entorno de programación común en Java. También presenta la tecnología de objetos, el Lenguaje de modelado unificado y los patrones de diseño.

Capítulo 2 "Introducción a las aplicaciones en Java": Ofrece una ligera introducción a la programación de *aplicaciones* en el lenguaje de programación Java. Este capítulo introduce los conceptos y construcciones básicas de programación a los no programadores. Los programas de este capítulo ilustran cómo mostrar datos en la pantalla para que los vea el usuario, y cómo obtener datos del usuario mediante el teclado. Parte de los procesos de entrada y salida se lleva a cabo mediante el uso de componentes de *interfaz gráfica de usuario (GUI)*. El capítulo 2 también ofrece un análisis detallado de la *toma de decisiones* y las *operaciones aritméticas*.

Capítulo 3 "Introducción a los subprogramas de Java": Presenta los *subprogramas* Java, que son programas en Java diseñados para transportarse a través de Internet y ejecutarse en navegadores Web (como Netscape Navigator y Microsoft Internet Explorer). Este capítulo muestra varios de los subprogramas de demostración que se incluyen con el J2SDK. Después escribiremos subprogramas de Java que llevan a cabo tareas similares a los programas del capítulo 2, y explicaremos las similitudes y diferencias entre los subprogramas y las aplicaciones.

Capítulo 4 "Instrucciones de control: Parte 1": Se enfoca en el proceso de desarrollo de programas. Este capítulo habla sobre cómo tomar un *enunciado del problema* y, a partir de éste, desarrollar un programa funcional en Java, incluyendo la realización de los pasos intermedios con *pseudocódigo*. También explica ciertos tipos primitivos de instrucciones simples de control para la toma de decisiones (`if` e `if...else`) y la repetición (`while`). También analizamos la repetición controlada por contador y la repetición controlada por centinela, y se introducen los operadores de incremento, decremento y asignación de Java. Este capítulo utiliza diagramas de actividad simples en UML para mostrar el flujo de control a través de cada una de las instrucciones de control.

Capítulo 5 "Instrucciones de control: Parte 2": Continúa hablando sobre las instrucciones de control de Java, con ejemplos de la instrucción de repetición `for`, la instrucción de repetición `do...while`, la instrucción de selección switch, las instrucciones break y continue. También se incluye una discusión sobre los operadores lógicos de Java.

Capítulo 6 "Métodos": Ofrece un análisis más detallado de los objetos. Los objetos contienen datos llamados campos y unidades ejecutables llamadas métodos. Aquí hablamos sobre los métodos de la biblioteca de

http://librosysolucionarios.net

clases y también creamos nuestros propios métodos. Para los cursos de ciencias computacionales, este capítulo presenta también una discusión sobre la recursividad. Las técnicas que se presentan en el capítulo 6 son esenciales para producir programas correctamente estructurados, especialmente los programas más grandes que desarrollan los programadores de sistemas y de aplicaciones. El tema de la sobrecarga de métodos (es decir, permitir que varios métodos tengan el mismo nombre, siempre y cuando tengan distintas "firmas") se motiva y se explica claramente. También presentamos los *eventos* y el *manejo de eventos*.

Capítulo 7 "Arreglos": Explora las listas de procesamiento y las tablas de valores. Los arreglos en Java son objetos, una evidencia más del compromiso de Java con la orientación a objetos. En este capítulo hablamos sobre la estructuración de datos en arreglos de elementos de datos del mismo tipo. Aquí también se presentan numerosos ejemplos, tanto de los arreglos unidimensionales como de los multidimensionales. Los ejemplos en este capítulo investigan las manipulaciones comunes de los arreglos, la impresión de histogramas, el proceso de pasar arreglos a los métodos y una introducción al campo del análisis de datos de encuestas (con estadísticas simples). Una característica de este capítulo es la discusión de las técnicas elementales para ordenar y realizar búsquedas, junto con la presentación de la búsqueda binaria como una mejora dramática sobre la búsqueda lineal.

Capítulo 8 "Programación basada en objetos": Empieza nuestra discusión más detallada acerca de las clases. Este capítulo representa una maravillosa oportunidad para enseñar la abstracción de datos de la "forma correcta"; a través de un lenguaje (Java) dedicado específicamente a la implementación de nuevos tipos. Se enfoca en la esencia y la terminología de las clases y los objetos. Habla sobre la implementación de clases en Java, el acceso a los miembros de una clase, cómo implementar la ocultación de la información con modificadores de acceso, separar la interfaz de la implementación, usar métodos de acceso y métodos utilitarios e inicializar objetos con constructores. El capítulo habla también sobre cómo declarar y utilizar constantes, la *composición*, la referencia con `this`, los miembros de clase `static` y ejemplos de tipos de datos abstractos populares, como las pilas y las colas. También introduce la instrucción package y habla sobre cómo crear paquetes reutilizables.

Capítulo 9 "Programación orientada a objetos: Herencia": En este capítulo se explica la herencia, una de las herramientas fundamentales de los lenguajes de programación orientados a objetos, la cual es una forma de reutilización de software en la que nuevas clases se desarrollan rápida y fácilmente al absorber las características de las clases existentes y al agregar nuevas características que sean apropiadas. También habla sobre las nociones de superclases y subclases, el modificador de acceso `protected`, las superclases directas e indirectas, el uso de constructores en superclases y subclases, y la ingeniería de software con la herencia. El capítulo compara la herencia (relaciones "es un") con la composición (relaciones "tiene un").

Capítulo 10 "Programación orientada a objetos: Polimorfismo": Este capítulo habla sobre otra herramienta fundamental de la programación orientada a objetos, mejor conocida como comportamiento polimórfico. Este estilo de programación se utiliza generalmente para implementar los populares marcos de trabajo de GUI, como Swing de Java. Este capítulo hace la distinción entre clases abstractas (`abstract`) y clases concretas, además de presentar las interfaces: el reemplazo de Java para la peligrosa (aunque poderosa) característica de C++ conocida como herencia múltiple. También presenta el poderoso concepto de *clases anidadas*, las cuales ayudan a ocultar los detalles de implementación. Luego, el capítulo demuestra nuestras primeras aplicaciones basadas en GUI, como parte de una introducción más completa al manejo de eventos. En esta sección utilizamos clases anidadas para definir los manejadores de eventos para varios componentes de GUI. Una característica notable de este capítulo es que cuenta con tres ejemplos prácticos de polimorfismo: un sistema de nóminas, una jerarquía de figuras encabezada por una clase `abstract` y una jerarquía de clases encabezada por una interfaz.

Capítulo 11 "Cadenas y caracteres": Este capítulo habla sobre el procesamiento de palabras, oraciones, caracteres y grupos de caracteres. Presentamos las clases String, `StringBuffer`, `Character` y `StringTokenizer`. También presentamos la API de Java para expresiones regulares (nueva en la J2SE 1.4), la cual permite a los programas buscar en las cadenas secuencias de caracteres que concuerden con ciertos patrones especificados.

Capítulo 12 "Gráficos y Java2D": Éste es el primero de varios capítulos en los que se presentan las herramientas gráficas y de multimedia de Java. Hablamos sobre los contextos y los objetos gráficos; el dibujo de cadenas, caracteres y bytes; el control de color y tipo de letra; la manipulación de la pantalla, los modos de tipo de talla y el dibujo de líneas, rectángulos, rectángulos redondeados, rectángulos tridimensionales, óvalos, arcos y

http://librosysolucionarios.net

polígonos. Presentamos la API Java2D, la cual ofrece poderosas herramientas para gráficos. La figura 12.22 es un ejemplo de lo fácil que es utilizar la API Java2D para crear efectos con gráficos complejos, como texturas y degradados.[5]

Capítulo 13 "Componentes de la interfaz gráfica de usuario, Parte 1": Este capítulo presenta varios de los *componentes Swing* de Java para crear programas con interfaces gráficas de usuario amigables (GUIs). Estos componentes de GUI *independientes de la plataforma* están escritos completamente en Java, lo que les proporciona una gran flexibilidad. Los componentes Swing pueden personalizarse para asemejarse a la apariencia de la plataforma computacional en la que se ejecute el programa, o pueden usar la apariencia visual estándar de Java para brindar una interfaz de usuario idéntica en todas las plataformas computacionales. El desarrollo de GUIs es un tema extenso, por lo que lo dividimos en dos capítulos. Estos capítulos cubren el material con suficiente detalle como para permitirle a usted crear interfaces de usuario atractivas y funcionales. Este capítulo ilustra los principios de las GUIs, la jerarquía `javax.swing`, las etiquetas, botones, listas, campos de texto, cuadros combinados, casillas de verificación, botones de opción, paneles, el manejo de eventos de ratón y de teclado, y los administradores de esquemas para posicionar componentes. El capítulo también hace énfasis en el manejo de eventos.

Capítulo 14 "Componentes de la interfaz gráfica de usuario: Parte 2": Este capítulo continúa con el tema sobre Swing que se trató en el capítulo 13. A través de sus programas, tablas y dibujos lineales, el capítulo ilustra los principios de diseño de GUIs, las áreas de texto, cómo extender los componentes Swing, los controles deslizables, las ventanas, los menús, menús contextuales, cómo cambiar la apariencia visual, las interfaces con documentos múltiples, los paneles con fichas y el uso de administradores de esquemas avanzados.[6]

Capítulo 15 "Manejo de excepciones": Éste es uno de los capítulos más importantes en el libro, desde el punto de vista de la creación de aplicaciones de "misión crítica" o de "comercio crítico". Los programadores necesitan hacerse cargo de "¿qué ocurre cuando el componente que llamo para hacer un trabajo presenta dificultades? ¿Cómo me indicará ese componente que tuvo un problema?" Para utilizar un componente de Java, es necesario saber no sólo la manera como se comporta ese componente cuando "las cosas salen bien", sino también qué *excepciones lanza* ese componente cuando "las cosas salen mal". Este capítulo hace la distinción entre los `errores` del sistema, que son bastante graves, y las `excepciones`. También habla sobre el vocabulario del manejo de excepciones, incluyendo los *bloques* `try`, *las cláusulas* `catch` y `finally`; además, introduce la nueva capacidad de encadenamiento de excepciones en la J2SE 1.4. El material de este capítulo es imprescindible para muchos de los ejemplos en el resto del libro.

Capítulo 16 "Subprocesamiento múltiple": Este capítulo habla sobre cómo desarrollar programas en Java que puedan llevar a cabo varias actividades en forma concurrente. Las computadoras solían construirse con un solo procesador, bastante costoso. Actualmente, los procesadores son cada vez más baratos, de manera que es posible construir computadoras con varios procesadores que trabajen en paralelo; dichas computadoras se conocen como *multiprocesadores*. La tendencia se inclina claramente hacia las computadoras que pueden realizar estas funciones. Como veremos, el subprocesamiento múltiple es efectivo aún en sistemas con un solo procesador. El capítulo presenta programas con subprocesamiento múltiple, los cuales muestran los problemas que pueden ocurrir en la programación concurrente. Una característica del capítulo es su extenso conjunto de ejemplos que muestran este tipo de problemas y cómo resolverlos. También se abordan los subprocesos y los métodos de los mismos. Avanza a través de los diversos estados de un subproceso y las transiciones de estado con una representación gráfica del ciclo de vida de un subproceso. Asimismo, discurre sobre las prioridades de los subprocesos y cómo programar su ejecución. Analiza una relación productor/consumidor sin sincronización, se observan los problemas que ocurren y se soluciona el problema con la sincronización de los subprocesos. Implementa una relación productor/consumidor con un búfer circular, y una sincronización adecuada con un monitor. Habla sobre los subprocesos tipo "demonio", que se la pasan "esperando" y llevan a cabo tareas cuando hay tiempo disponible en el procesador. Finalmente, otro punto que se trata es sobre la interfaz `Runnable`, que permite a los objetos ejecutarse como subprocesos sin tener que usar la clase `Thread` como subclase.

---

5. Nuestro libro complementario, *Advanced Java 2 Platform How to Program*, presenta la API 3D, útil para crear efectos tridimensionales.
6. Los capítulos 1 a 6 de *Advanced Java 2 Platform How to Program*, ofrece conceptos de la GUI más avanzados.

Capítulo 17 "Archivos y flujos": Este capítulo trata acerca de la entrada/salida que se logra a través de flujos de datos dirigidos hacia, y provenientes desde, los archivos. En este capítulo se traducen los objetos a un formato persistente. El proceso de poder almacenar datos en archivos o desplazarlos a través de redes (capítulo 18) hace que los programas puedan guardar datos y comunicarse entre sí. Este capítulo empieza con una introducción a la jerarquía de datos, comenzando desde los bits, después los bytes, los campos, los registros y los archivos. A continuación se presenta un análisis simple en Java de los archivos y flujos. Mostramos cómo los programas pasan datos a los dispositivos de almacenamiento secundario como los discos, y la forma en que recuperan los datos almacenados en esos dispositivos. También hay una discusión sobre la clase `File`, que es utilizada por los programas para obtener información acerca de archivos y directorios. Se explica el método por el cual los objetos pueden enviarse hacia, y recibirse desde, los dispositivos de almacenamiento secundario. También hay un apartado sobre las Nuevas APIs de E/S de alto rendimiento (introducidas en la J2SE 1.4).

Capítulo 18 "Redes": Este capítulo habla sobre los programas en Java que se comunican a través de redes de computadoras. Presenta las herramientas de red de menor nivel en Java. Los ejemplos de este capítulo muestran a un subprograma interactuando con el navegador en el que se ejecuta, creando un mini navegador Web, comunicándose entre dos programas en Java mediante sockets basados en flujos, y comunicándose entre dos programas en Java mediante paquetes de datos. Una característica clave de este capítulo es la implementación de un juego de Tres en línea que utiliza la arquitectura cliente/servidor en colaboración, en donde dos clientes juegan Tres en línea uno con el otro, y el juego es arbitrado por un servidor con subprocesamiento múltiple. ¡Estupendo! El ejemplo culminante en este capítulo es el del Deitel Messenger, que simula a muchas de las aplicaciones de mensajería instantánea populares hoy en día, las cuales permiten a los usuarios de computadora comunicarse con amigos, parientes y compañeros de trabajo a través de Internet. Este ejemplo práctico de cliente/servidor, con 1,130 líneas y subprocesamiento múltiple, utiliza la mayoría de las técnicas de programación presentadas hasta este punto. Este capítulo también continúa con el tema sobre las nuevas APIs de E/S de la J2SE 1.4, con una introducción a los selectores y la E/S sin bloqueo, para implementar servidores de red con alto rendimiento.[7]

Capítulo 19 "Multimedia: imágenes, animación y audio": Presenta algunas de las herramientas de Java para hacer que los programas cobren vida a través de la multimedia. Este capítulo habla sobre las imágenes y la manipulación de éstas, el audio y la animación. También presenta una aplicación LIVE-CODE de mapa de imágenes, con los iconos de los tips de programación que se mostraron antes en este prefacio, y que aparecen en todo el libro. A medida que el usuario mueve el puntero del ratón por los iconos, se va mostrando el tipo de tip. Una vez que haya leído el capítulo, estará ansioso por probar todas estas técnicas, por lo que se han incluido muchos ejercicios para retarlo y entretenerlo.

Capítulo 20 "Estructuras de datos": Es especialmente valioso en cursos superiores de segundo y tercer nivel. Este capítulo versa sobre las técnicas utilizadas para crear y manipular estructuras dinámicas de datos, como listas enlazadas, pilas, colas y árboles. Para cada tipo de estructura de datos, se presentan ejemplos con muestras de salida. Aunque es valioso saber cómo se implementan estas clases, los programadores de Java descubrirán rápidamente que la mayoría de las estructuras de datos que necesitan están disponibles en las bibliotecas de clases, como la biblioteca `java.util` de Java que describimos en los capítulos 21 y 22.

Capítulo 21 "Paquete de utilerías de Java y manipulación de bits": Presenta varias clases de la biblioteca `java.util` y habla sobre los operadores de manipulación de bits de Java. Una clase especialmente útil es `Vector`: un arreglo dinámico que puede crecer y reducirse según sea necesario. También hablamos sobre las clases `Stack`, `Hashtable`, `Properties`, `Random` y `BitSet`.

Capítulo 22 "Colecciones": Habla sobre las clases de `java.util` que pertenecen a la API Collections, y que ofrecen implementaciones predefinidas de muchas de las estructuras de datos descritas en el capítulo 20. Las colecciones ofrecen a los programadores de Java un conjunto estándar de estructuras de datos para almacenar y recuperar datos, y un conjunto estándar de algoritmos (es decir, procedimientos) que permiten a los programadores manipular esos datos (como buscar ciertos elementos de datos y ordenar los datos en forma ascendente o descendente). Este capítulo demuestra las colecciones tales como listas enlazadas, árboles, mapas y conjuntos, y los algoritmos para buscar, ordenar, encontrar el valor máximo, el mínimo y así, sucesivamente.

7. Nuestro libro complementario, *Advanced Java 2 Platform How to Program*, ofrece un análisis mucho más profundo de las redes y la computación distribuida, con temas que incluyen la invocación a métodos remotos (RMI), Java 2 Enterprise Edition, Java inalámbrico (y Java 2 Micro Edition) y CORBA (Common Object Request Broker Architecture, Arquitectura común de agente de solicitud de objetos).

Capítulo 23 "Conectividad de bases de datos en Java con JDBC™": Habla sobre el soporte que ofrece Java para usar bases de datos. Los sistemas de bases de datos más populares hoy en día son las bases de datos relacionales. Presentamos ejemplos utilizando Cloudscape de IBM, un sistema de administración de bases de datos creado completamente en Java. Este capítulo introduce la JDBC, utilizándola para conectarse con una base de datos Cloudscape y después manipular su contenido. Utilizamos el Lenguaje de consulta estructurado (SQL) para extraer información de, e insertar información en, una base de datos. Los siguientes capítulos sobre servlets y JavaServer Pages utilizan las técnicas que se muestran en este capítulo para crear aplicaciones Web controladas por datos.

Capítulo 24 "Servlets": Habla sobre los servlets, que generalmente extienden la funcionalidad de los servidores Web. Los servlets son efectivos para desarrollar soluciones basadas en Web que interactúan con bases de datos a beneficio de los clientes, generan contenido personalizado en forma dinámica para que lo muestren los navegadores, y mantienen información de sesión única por cada cliente. La API para servlets de Java permite a los desarrolladores aumentar la funcionalidad de los servidores Web para hacerse cargo de las peticiones de los clientes. Los servlets también pueden reutilizarse en la mayoría de los servidores Web y plataformas. Este capítulo muestra el mecanismo de petición/respuesta de Web (principalmente con las peticiones `get` y `post` de HTTP), redireccionando las peticiones hacia otros recursos e interactuando con las bases de datos a través de JDBC. El capítulo presenta una aplicación cliente/servidor de tres niveles que da seguimiento a las respuestas de un usuario en una encuesta.

Capítulo 25 "JavaServer Pages (JSP)": Presenta una extensión de la tecnología de servlets conocida como JavaServer Pages (JSP). Las JSPs permiten el uso de contenido Web generado en forma dinámica, y son utilizadas principalmente por los diseñadores Web y todos los que no están familiarizados con la programación en Java. Las JSPs pueden contener código de Java en forma de scriptlets. Para aumentar el rendimiento, cada JSP se compila en un servlet de Java; esto normalmente ocurre la primera vez que un cliente solicita la JSP. Las peticiones subsecuentes de los clientes son atendidas por el servlet compilado. Este capítulo presenta una aplicación de libro de visitantes, tipo cliente/servidor de tres niveles, que almacena la información de los visitantes en una base de datos.

Apéndice A "Tabla de precedencia de operadores": Enlista cada uno de los operadores de Java e indica su precedencia relativa y su asociatividad.

Apéndice B "Conjunto de caracteres ASCII": Enlista los caracteres del conjunto ASCII (Código estándar estadounidense para el intercambio de información) e indica el valor para cada uno de ellos en código de carácter. Java utiliza el conjunto de caracteres Unicode, con caracteres de 16 bits, para representar todos los caracteres en los lenguajes "comercialmente considerables" en el mundo. Unicode incluye el código ASCII como un subconjunto.

Apéndice C "Sistemas numéricos": Habla sobre los sistemas numéricos binario (base 2), decimal (base 10), octal (base 8) y hexadecimal (base 16). Este material es valioso para los cursos introductorios en ciencias computacionales e ingeniería computacional.

Apéndices D al F: Contienen la implementación de nuestro ejemplo práctico sobre Diseño orientado a objetos con UML. Éste se discute en la descripción general del ejemplo práctico.

Apéndice G "Unicode": Habla sobre el conjunto de caracteres Unicode, el cual permite a Java mostrar la información en muchos lenguajes. Este apéndice incluye un programa Java de ejemplo que muestra "Bienvenido a Unicode" en varios lenguajes.

(Opcional) Un paseo por el ejemplo práctico sobre Diseño orientado a objetos con UML

En esta sección y en la siguiente daremos un paseo a través de las dos principales características opcionales del libro: el ejemplo práctico opcional de diseño orientado a objetos con UML y nuestra introducción a los patrones de diseño. El ejemplo práctico es una adición importante a *Cómo programar en Java, 5ª edición*. Este paseo presenta un avance del contenido de las secciones "Acerca de los objetos" y habla sobre cómo se relacionan con el ejemplo práctico. Después de realizar este ejemplo práctico, usted habrá completado el diseño orientado a objetos y la implementación para una aplicación importante en Java.

http://librosysolucionarios.net

Sección 1.15, Acerca de los objetos: Introducción a la tecnología de objetos y el Lenguaje de modelado unificado

Esta sección introduce el ejemplo práctico de diseño orientado a objetos con UML. Aquí presentamos los antecedentes generales de lo que son los objetos y cómo interactúan entre sí. También hablamos brevemente sobre el estado de la industria de la ingeniería de software y la forma en que UML ha influenciado los procesos de análisis y diseño orientado a objetos.

Sección 2.9 (Ejemplo práctico opcional), Acerca de los objetos: Cómo examinar el enunciado de un problema

Nuestro ejemplo práctico comienza con un *enunciado de problema*, el cual especifica los requerimientos para el sistema que vamos a crear. En este ejemplo práctico estudiamos e implementamos la simulación de un sistema de elevador en un edificio de dos plantas. Proporcionamos el diseño de nuestro sistema de elevador después de investigar la estructura y el comportamiento de los sistemas orientados a objetos en general. Hablamos sobre cómo UML facilita el proceso de diseño en las siguientes secciones "Acerca de los objetos", al ofrecernos varios tipos de diagramas para modelar nuestro sistema. Por último, proporcionamos una lista de referencias de URLs y libros que tratan sobre el diseño orientado a objetos con UML. Tal vez le sean útiles estas referencias a medida que progrese a través de la presentación de nuestro ejemplo práctico.

Sección 3.7 (Ejemplo práctico opcional), Acerca de los objetos: Cómo identificar las clases en el enunciado de un problema

En esta sección empezamos a diseñar la simulación del elevador. Identificamos las clases o "bloques de construcción" de nuestra simulación, extrayendo los sustantivos y las frases nominales del enunciado del problema. Ordenamos estas clases en un diagrama de clases de UML, el cual describe la estructura de clases de nuestra simulación. El diagrama de clases también describe las relaciones, conocidas como *asociaciones*, entre las clases (por ejemplo, una persona tiene una asociación con el elevador, ya que ésta viaja en él).

Sección 4.14 (Ejemplo práctico opcional), Acerca de los objetos: Cómo identificar los atributos de una clase

Una clase contiene tanto *atributos* (datos) como *operadores* (comportamientos). Esta sección se enfoca en los atributos de las clases que se describieron en la sección 3.7. Como veremos en secciones posteriores, los cambios en los atributos de un objeto a menudo afectan al comportamiento de ese objeto. Para determinar los atributos para las clases en nuestro ejemplo práctico, extraemos los adjetivos que describen a los sustantivos y frases nominales (que definen a nuestras clases) del enunciado del problema y después colocaremos los atributos en el diagrama de clases que creamos en la sección 3.7.

Sección 5.11 (Ejemplo práctico opcional), Acerca de los objetos: Cómo identificar los estados y actividades de los objetos

Un objeto, en cualquier momento dado, ocupa una condición específica conocida como *estado*. Una *transición de estado* ocurre cuando ese objeto recibe un mensaje para cambiar de estado. UML cuenta con el *diagrama de estados*, el cual identifica el conjunto de posibles estados que puede ocupar un objeto y modela las transiciones de estado de ese objeto. Un objeto tiene también una *actividad*: el trabajo desempeñado por un objeto durante su vida. UML cuenta con el *diagrama de actividades*: un diagrama de flujo que modela la *actividad* de un objeto. En esta sección utilizamos ambos tipos de diagramas para empezar a modelar ciertos aspectos específicos del comportamiento de nuestra simulación de elevador, como la forma en que una persona viaja en el elevador y cómo éste responde cuando se oprime un botón en un piso dado.

Sección 6.15 (Ejemplo práctico opcional), Acerca de los objetos: Cómo identificar las operaciones de una clase

En esta sección identificamos las operaciones, o servicios, de nuestras clases. Extrajimos del enunciado del problema los verbos y frases verbales que especifican las operaciones para cada clase. Después modificamos el diagrama de clases de la sección 3.7 para incluir cada operación con su clase asociada. En este punto del ejemplo práctico, hemos recopilado toda la información posible del enunciado del problema. Sin embargo, y a medida que los siguientes capítulos introduzcan temas como la herencia, el manejo de eventos y el subprocesamiento múltiple, modificaremos nuestras clases y diagramas.

http://librosysolucionarios.net

Sección 7.10 (Ejemplo práctico opcional), Acerca de los objetos: Colaboración entre objetos

En este punto hemos creado un "borrador" del modelo para nuestro sistema de elevador. En esta sección vemos cómo funciona. Investigamos el comportamiento de la simulación al hablar sobre las *colaboraciones* (mensajes que los objetos se envían entre sí para comunicarse). Las operaciones de las clases que hemos descubierto en la sección 6.15 resultan ser las colaboraciones entre los objetos en nuestro sistema. Determinamos las colaboraciones en nuestro sistema y después las reunimos en un *diagrama de colaboración*: el diagrama de UML para modelar colaboraciones. Este diagrama revela qué objetos colaboran y cuándo lo hacen. Presentamos un diagrama de colaboración de las personas que entran y salen del elevador.

Sección 8.17 (Ejemplo práctico opcional), Acerca de los objetos: Cómo empezar a programar las clases para la simulación del elevador

En esta sección tomamos un descanso, dejando a un lado el diseño del comportamiento de nuestro sistema. Comenzamos el proceso de implementación para enfatizar el material descrito en el capítulo 8. Utilizando el diagrama de clases de UML de la sección 3.7 y los atributos y operaciones descritas en las secciones 4.14 y 6.15, mostramos cómo implementar una clase en Java, partiendo de un diseño. No implementamos todas las clases, ya que no hemos completado el proceso de diseño. Basándonos en nuestros diagramas de UML, creamos código para la clase `Elevador`.

Sección 10.12 (Ejemplo práctico opcional), Acerca de los objetos: Cómo incorporar la herencia a la simulación del elevador

Los capítulos 9 y 10 hablan sobre la programación orientada a objetos. Consideramos la herencia: las clases que comparten características similares pueden heredar los atributos y operaciones de una clase "base". En esta sección investigamos cómo nuestra simulación del elevador puede beneficiarse a partir del uso de la herencia. Documentamos nuestros descubrimientos en un diagrama de clases que modela las relaciones de herencia; UML se refiere a estas relaciones como *generalizaciones*. Después modificamos el diagrama de clases de la sección 3.7 mediante el uso de la herencia, para agrupar las clases con características similares.

Sección 11.9 (Ejemplo práctico opcional), Acerca de los objetos: Manejo de eventos

En esta sección incluimos las interfaces necesarias para que los objetos en nuestra simulación del elevador envíen mensajes a otros objetos. En Java, los objetos se comunican generalmente enviando un *evento*: una notificación de que ha ocurrido una acción. El objeto que recibe el evento entonces realiza una acción en respuesta al tipo de evento recibido; esto se conoce como *manejo de eventos*. En la sección 7.10 describimos el paso de mensajes (o las colaboraciones) en nuestro modelo, utilizando un diagrama de colaboraciones. Ahora modificamos este diagrama para incluir el manejo de eventos y, como ejemplo, explicamos detalladamente cómo las puertas en nuestra simulación se abren al momento de la llegada del elevador.

Sección 12.9 (Ejemplo práctico opcional), Acerca de los objetos: Diseño de interfaces con UML

En esta sección diseñamos un diagrama de clases que modela las relaciones entre clases e interfaces en nuestra simulación; UML se refiere a estas relaciones como *realizaciones*. Además, enlistamos todas las operaciones que proporciona cada interfaz a las clases. Por último, mostramos cómo crear clases en Java que implementen estas interfaces.

Sección 13.17 (Ejemplo práctico opcional), Acerca de los objetos: Casos de uso

El capítulo 13 habla sobre las interfaces de usuario que le permiten interactuar con un programa. En esta sección hablamos sobre la interacción entre nuestra simulación del elevador y su usuario. Específicamente, investigamos los casos que pueden ocurrir entre el usuario de la aplicación y la simulación en sí; este conjunto de casos se conoce como *caso de uso*. Modelamos estas interacciones utilizando *diagramas caso de uso* de UML.

Sección 14.13 (Ejemplo práctico opcional), Acerca de los objetos: Modelo-vista-controlador

Diseñamos nuestro sistema para que consista de tres componentes, cada uno de los cuales tiene una responsabilidad distinta. En este punto del ejemplo práctico casi hemos completado el primer componente, conocido como *modelo*, que contiene datos que representan a la simulación. Diseñaremos la *vista* (el segundo componente, que trata acerca de la manera en que se muestra el modelo) en la sección 19.7. Diseñaremos el *contro-*

lador (el componente que permite al usuario controlar el modelo) en esta sección. Se dice que un sistema como el nuestro, que utiliza los componentes modelo, vista y controlador, se adhiere a la arquitectura *Modelo-vista-controlador* (*MVC*). En esta sección explicamos las ventajas de utilizar esta arquitectura para diseñar software. Utilizamos el *diagrama de componentes* de UML para modelar los tres componentes y luego implementamos este diagrama como código en Java.

Sección 16.11 (Ejemplo práctico opcional), Acerca de los objetos: Subprocesamiento múltiple
En esta sección declaramos ciertos objetos como "subprocesos" para que éstos puedan operar en forma concurrente. Modificamos el diagrama de colaboración que se presentó originalmente en la sección 7.10 (y que se modificó en la sección 11.9) para incorporar el subprocesamiento múltiple. Presentamos el *diagrama de secuencia* de UML para modelar las interacciones en un sistema. Este diagrama enfatiza el orden cronológico de los mensajes. Utilizamos un diagrama de secuencia para modelar la forma en que una persona dentro de la simulación interactúa con el elevador. Esta sección concluye el diseño de la porción de nuestra simulación relacionada con el modelo. Diseñamos la manera en que se muestra este modelo en la sección 19.7 y luego implementamos este modelo como código en Java, en el apéndice E.

Sección 19.7 (Ejemplo práctico opcional), Acerca de los objetos: Animación y sonido en la vista
Esta sección diseña la vista, que especifica cómo se muestra la porción de la simulación relacionada con el modelo. El capítulo 19 presenta varias técnicas para integrar sonido y animación en los programas. Esta sección utiliza algunas de estas técnicas para incorporar sonido y animación en nuestra simulación del elevador. Específicamente, habla sobre cómo animar los movimientos de las personas y de nuestro elevador, cómo generar efectos y tocar "música de elevador" cuando una persona viaja en él. Esta sección concluye el diseño de nuestra simulación de elevador. Los apéndices D, E y F implementan este diseño como un programa en Java de 3,320 líneas, completamente funcional.

Apéndice D: Eventos e interfaces de escucha del elevador
[Nota: Este apéndice se encuentra en el CD que viene con este libro.] Como vimos en la sección 11.9, varios objetos en nuestra simulación interactúan entre sí mediante el envío de mensajes, conocidos como eventos, a otros objetos que desean recibir estos eventos. Los objetos que reciben los eventos se conocen como *objetos de escucha*; éstos deben implementar *interfaces de escucha*. En este apéndice implementamos todas las clases de eventos e interfaces de escucha utilizadas por los objetos en nuestra simulación.

Apéndice E: Modelo del elevador
[Nota: Este apéndice se encuentra en el CD que viene con este libro.] La mayor parte del ejemplo práctico involucra el diseño del modelo (es decir, los datos y la lógica) de la simulación del elevador. En este apéndice implementamos ese modelo en Java. Utilizando todos los diagramas de UML que creamos, presentamos las clases de Java necesarias para implementar el modelo. Aplicamos los conceptos del diseño orientado a objetos con UML y la programación orientada a objetos, y con el nivel de Java que usted aprendió en los capítulos.

Apéndice F: Vista del elevador
[Nota: Este apéndice se encuentra en el CD que viene con este libro.] Este apéndice final implementa la apariencia visual de nuestra simulación del elevador. Utilizamos el mismo enfoque para implementar la vista que para implementar el modelo; creamos todas las clases requeridas para ejecutar la vista, utilizando los diagramas de UML y conceptos clave descritos en los capítulos. Al final de este apéndice, usted habrá completado el diseño y la implementación "a nivel industrial" de un sistema a gran escala. Debe sentirse con la confianza de hacer frente a sistemas más extensos, como el ejemplo práctico empresarial en Java de 10,000 líneas que presentamos en nuestro libro complementario *Advanced Java 2 Platform How to Program* y los tipos de aplicaciones que crean los ingenieros de software profesionales. Esperamos que continúe estudiando aún con más profundidad el diseño orientado a objetos con UML.

(Opcional) Un paseo por las secciones "Descubrimiento de patrones de diseño"

Nuestro análisis de los patrones de diseño se esparce a lo largo de cinco secciones opcionales del libro. Aquí veremos las generalidades acerca de esas secciones.

Sección 10.12 (Opcional), Descubrimiento de los patrones de diseño: Introducción a los patrones de diseño de creación, estructura y comportamiento
Aquí encontrará las secciones en las que hablamos sobre los diversos patrones de diseño. Dividimos la discusión de cada sección en patrones de diseño de creación, estructura y comportamiento. Los patrones de creación proporcionan maneras de crear instancias de objetos, los patrones de estructura tratan con la organización de los objetos y los patrones de comportamiento tratan con las interacciones entre los objetos. El resto de esta sección introduce algunos de estos patrones de diseño como Singular (Singleton), Proxy, Memento y Estado (State). Por último incluimos varios URLs para profundizar más en el estudio de los patrones de diseño.

Sección 14.14 (Opcional), Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.awt` y `javax.swing`
Esta sección contiene una buena parte de nuestra discusión sobre los patrones de diseño. Usando el material de los componentes GUI Swing de Java en los capítulos 13 y 14, investigamos algunos ejemplos del uso de patrones en los paquetes `java.awt` y `javax.swing`. Hablamos sobre cómo estas clases utilizan los patrones de diseño Método de fábrica (Factory Method), Adaptador (Adapter), Puente (Bridge), Compuesto (Composite), Cadena de responsabilidad (Chain-of-Responsibility), Comando (Command), Observador (Observer), Estrategia (Strategy) y Método de plantilla (Template Method). Motivamos cada patrón y presentamos ejemplos de cómo aplicarlos.

Sección 16.12 (Opcional), Descubrimiento de los patrones de diseño: Patrones de diseño concurrentes
Los desarrolladores han descubierto varios patrones de diseño desde aquellos que se conocían como la banda de los cuatro. En esta sección hablamos sobre los patrones de diseño de concurrencia, incluyendo: Ejecución sde un solo subproceso (Single-Threaded Execution), Suspensión precavida (Guarded Suspension), Frustración (Balking), Bloqueo de lectura/escritura (Read/Write Lock) y Término en dos fases (Two-Phase Termination); éstos resuelven varios problemas de diseño en los sistemas con subprocesamiento múltiple. Investigamos cómo la clase `java.lang.Thread` utiliza los patrones de concurrencia.

Sección 18.12 (Opcional), Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.io` y `java.net`
Usando el material sobre archivos, flujos y redes de los capítulos 17 y 18, investigamos algunos ejemplos del uso de patrones en los paquetes `java.io` y `java.net`. Hablamos sobre cómo estas clases utilizan los patrones de diseño Fábrica abstracta (Abstract Factory), Decorador (Decorator) y Fachada (Facade). También consideramos los patrones de arquitectura, que especifican un conjunto de subsistemas (agregados de objetos que en conjunto abarcan una responsabilidad importante del sistema) y cómo estos subsistemas interactúan unos con otros. Hablamos sobre los populares patrones arquitectónicos Modelo-vista-controlador y Niveles (Layers).

Sección 22.12 (Opcional), Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en el paquete `java.util`
Utilizando el material sobre estructuras de datos y colecciones en los capítulos 20 al 22, investigamos el uso de patrones en el paquete `java.util`. Hablamos sobre cómo estas clases utilizan los patrones de diseño Prototipo (Prototype) e Iterador (Iterator). Esta sección concluye la discusión sobre los patrones de diseño. Después de terminar el material de *Descubrimiento de los patrones de diseño*, usted deberá ser capaz de reconocer y utilizar los patrones de diseño claves y de tener una mejor comprensión sobre el funcionamiento de la API de Java. Después de completar este material, le recomendamos que siga con el libro de la Banda de los cuatro (Gang-of-Four).

Reconocimientos

Uno de los mayores placeres al escribir un libro de texto es el de reconocer el esfuerzo de mucha gente, cuyos nombres quizá no aparezcan en la portada, pero cuyo arduo trabajo, cooperación, amistad y comprensión fue crucial para la elaboración de este libro.

Mucha gente en Deitel & Associates, Inc. dedicó largas horas a este proyecto. Nos gustaría reconocer los esfuerzos de nuestros colegas de tiempo completo en Deitel & Associates, Inc.: Tem Nieto, Sean Santry, Su Zhang y Jeff Listfield.

http://librosysolucionarios.net

- Tem Nieto es egresado del Massachusetts Institute of Technology. Imparte seminarios de XML, Java, Internet y Web, C, C++ y Visual Basic, y trabaja con nosotros en la escritura de libros de texto, el desarrollo de cursos y creación de multimedia. Es coautor de varios libros con nosotros, incluyendo *Internet & World Wide Web How to Program (segunda edición), XML How to Program, Visual Basic .NET How to Program* y *C# How to Program*. En este libro, Tem participó como coautor en los capítulos 12 al 14 y 22, y en la sección especial titulada "Construya su propio compilador" en el capítulo 20.
- Sean Santry, egresado del Boston College (Ciencias computacionales y Filosofía) y coautor de *Advanced Java 2 Platform How to Program* y *Java Web Services for Experienced Programmers*, editó todo el manuscrito, diseñó e implementó la aplicación de red Deitel Messenger en el capítulo 18 (Redes), contribuyó en el diseño y actualizó el ejemplo práctico opcional sobre DOO/UML, y actualizó la introducción opcional a los patrones de diseño.
- Su Zhang posee títulos B.Sc y M.Sc en Ciencias computacionales de la Universidad McGill. Su tesis incluyó el modelado y la simulación, los sistemas de tiempo real y la tecnología Java. Ella es coautora con nosotros en *Java Web Services for Experienced Programmers* y ha contribuido en otras publicaciones Deitel, incluyendo *Advanced Java 2 Platform How to Program* y *Python How to Program*. Ayudó a actualizar varios capítulos y creó los ejemplos que presentan nuevas características de Java 1.4, como las Nuevas APIs de E/S (de las cuales se habla en los capítulos 17 y 18) y las excepciones encadenadas (que se mencionan en el capítulo 15).
- Jeff Listfield es egresado en Ciencias computacionales del Harvard College. Su trabajo escolar incluyó clases en gráficos computacionales, redes y teoría computacional, y tiene experiencia de programación en C, C++, C#, Java, Perl y Lisp. Jeff es nuestro coautor en *C# How to Program*, *C# A Programmer's Introduction* y *C# for Experienced Programmers*, además contribuyó en *Perl How to Program*. Jeff ayudó a actualizar varios capítulos en el libro, escribió nuevos ejemplos y secciones sobre expresiones regulares en el capítulo 11, ordenamiento en el capítulo 22 y partes de las secciones sobre la Nueva API de E/S en los capítulos 17 y 18).

Somos afortunados al haber trabajado en este proyecto con un talentoso y dedicado equipo de editores profesionales. Apreciamos de manera especial el extraordinario esfuerzo de nuestra editora de Ciencias computacionales, Petra Recter y su jefa, nuestra mentora en la edición, Marcia Horton, directora editorial de la división de Ciencias de la computación e ingeniería de Prentice Hall. Tom Manshreck hizo un estupendo trabajo como gerente de producción, Jennifer Cappello hizo un excelente trabajo en el manejo del proceso de revisión.

El *Java 2 Multimedia Cyber Classroom, 5ª edición* se desarrolló en paralelo con *Cómo programar en Java, 5ª edición*. Sinceramente apreciamos la perspicacia, experiencia y habilidad técnica de nuestro editor en jefe de medios electrónicos, mentor y amigo, Mark Taub. Él y nuestra editora de medios electrónicos, Karen Mclean, hicieron un gran trabajo al lograr la publicación de *Java 2 Multimedia Cyber Classroom, 5ª edición* bajo un estrecho itinerario.

Debemos un agradecimiento especial a Tamara Newnam (`smart_art@earthlink.net`), quien creó el trabajo artístico para nuestros iconos de tips de programación y la portada. Ella creó la encantadora criatura que comparte con usted los tips de programación del libro.

Sinceramente apreciamos los esfuerzos de nuestros 70 revisores de la cuarta edición después de su publicación, y de nuestros revisores de la quinta edición:

Revisores de Sun Microsystems
Dibyendu Baksi (Sun Microsystems)
Konstantin Kladko (Sun Microsystems)
Doug Kohlert (Sun Microsystems)
Peter Jones (Sun Microsystems)
Paul Monday (Sun Microsystems)
Tomas Pavek (Sun Microsystems)
Brandon Taylor (Sun Microsystems)

Revisores académicos
Rekha Bhowmik (St. Cloud State University)

http://librosysolucionarios.net

Clint Bickmore (Front Range Community College)
Brian Blake (Georgetown University)
Ayad Boudiab (Georgia Perimiter College)
Chadi Boudiab (Georgia Perimiter College)
Michael Buckley (State University of New York-Buffalo)
James Chegwidden (Tarrant County College)
Deborah Coleman (Rochester Institute of Technology)
Don Francis Costello (University of Nebraska)
Balazs Csizmazia (University of Klagenfurt)
Tamara Dinev (Florida Atlantic University)
Sarah Fix (The Career Center High School)
Bill Freitas (The Lawrenceville School)
Thomas Graffte (Strayer University)
Balaji Janamanchi (Texas Tech University)
Charles Lake (Faulkner State Community College)
Brian Larson (Modesto Junior College)
Hong Lin (DeVry University)
David McKain (Lakota East High School)
Andy Novobilski (University of Tennessee-Chattanooga)
Richard Ord (University of California, San Diego)
Gavin Osborne (Saskatchewan Institute of Applied Sci.& Tech.)
Merrill Parker (Chattanooga State Technical Community College)
Donna Reese (Mississippi State University)
Craig Slinkman (University of Texas, Arlington)
Gidget Smith (Arkansas State University)
Ron Sones (James Madison University)
Mahendran Velauthapillai (Georgetown University)
Loran Walker (Lawrence Technological University)
Warren Wiltsie (Fairleigh Dickinson University)

Otros revisores de la industria
Shishir Abhyanker (Accenture)
Sinan Alhir (Consultor independiente)
Richard Bonneau (IONA Technologies)
Columbus Brown (IBM)
Carl Burnham (Southpoint)
Brian Cook (Zurich Insurance)
Jonathan Earl (Consultor independiente)
Ron Felice (Omniware Development)
Karl Frank (TogetherSoft Corporation)
Kyle Gabhart (Consultor independiente)
Johan Galle (E2S)
Mark Grand (ClickBlocks, LLC)
Ajay Gupta (American Airlines, Inc.)
Kevlin Henney (Curbralan, Ltd.)
Ethan Henry (Sitraka)
Anne Horton (AT&T Laboratories)
James Huddleston (Consultor independiente)
Terrell Hull (Sun Certified Java Architect, Rational Qualified Practitioner)
Sachin Korgaonkar (Idealake Technologies, Pvt. Ltd.)
Don Kostuch (You Can C Clearly Now)
Krishna Kunchithapadam (Oracle Corporation)

Paul McLachlan (Compuware Corporation)
Davyd Norris (Rational)
Bill O'Farrell (IBM)
Praveen Sadhu (Infodat Solutions, Inc.)
Cameron Skinner (Embarcadero Technologies/OMG)
Stephen Tockey (Construx Software/OMG)
Kim Topley (Keyboard Edge, Ltd.)
Sudhir Upadhyay (BEA Systems, Inc.)
John Varghese (UBS Warburg)
Bing Xue (Siemens)
Hadar Ziv (eBuilt, Inc.)

A pesar de una calendarización muy apretada, estos revisores revisaron cada aspecto del libro e hicieron incontables sugerencias para mejorar la precisión e integridad del material presentado.

Hemos hecho un gran esfuerzo para crear este libro y su versión opcional de Cyber Classroom. Esta edición está repleta de ejemplos LIVE-CODE™, tips de programación, ejercicios de autoevaluación con sus respuestas, ejercicios y proyectos retadores, y numerosas herramientas de estudio para ayudarlo a dominar el material. Java es un poderoso lenguaje de programación que le ayudará a escribir programas con rapidez y efectividad. Además, es un lenguaje que escala sin problemas en el ámbito del desarrollo de sistemas empresariales, para ayudar a las organizaciones a crear sus sistemas de información claves. Le agradeceremos que, a medida que lea el libro, nos haga saber sus comentarios, críticas, correcciones y sugerencias para mejorar el texto. Por favor dirija toda su correspondencia a:

`deitel@deitel.com`

Le responderemos oportunamente y publicaremos las correcciones y aclaraciones en nuestro sitio Web,

`www.deitel.com`

¡Esperamos que disfrute aprendiendo con este libro tanto como nosotros disfrutamos el escribirlo!

Dr. Harvey M. Deitel
Paul J. Deitel

Acerca de los autores

Dr. Harvey M. Deitel. Presidente de Deitel & Associates, Inc., tiene más de 40 años de experiencia en el campo de la computación, esto incluye un amplio trabajo académico y en la industria. Es uno de los profesores de ciencias computacionales y expositores más reconocidos a nivel mundial. El Dr. Deitel tiene una licenciatura y una maestría por el Massachusetts Institute of Technology y un doctorado de la Boston University. Tiene 20 años de experiencia como profesor universitario, incluyendo un puesto vitalicio y el haber sido presidente del departamento de Ciencias computacionales en el Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Es autor y coautor de varias docenas de libros y paquetes multimedia. Los textos del Dr. Deitel se han ganado el reconocimiento internacional y han sido traducidas al japonés, ruso, español, italiano, chino tradicional, coreano, francés, polaco, turco, urdú, griego y portugués. El Dr. Deitel ha impartido seminarios profesionales internacionalmente para grandes empresas, organizaciones gubernamentales y diversos sectores del ejército.

Paul J. Deitel. CEO de Deitel & Associates, Inc., es egresado del Sloan School of Management del Massachusetts Institute of Technology, en donde estudió Tecnología de la información. A través de Deitel & Associates, Inc., ha impartido seminarios de lenguajes computacionales a grandes empresas, organizaciones gubernamentales y diversos sectores del ejército. Ha ofrecido conferencias de Java y C++ para la Boston Chapter de la Association for Computing Machinery, y ha impartido cursos vía satélite a través de una empresa de cooperación entre Deitel & Associates, Inc., Prentice Hall y Technology Education Network. Él y su padre, Dr. Harvey M. Deitel, son autores de los libros de Ciencias de la computación con más ventas en el mundo.

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc. es una empresa reconocida a nivel mundial, dedicada al entrenamiento corporativo y la creación de contenido, con especialización en educación para tecnología de software para Internet/World Wide Web, tecnología de software para comercio electrónico (e-business/e-commerce) y lenguajes de programación. La empresa proporciona cursos sobre programación en Internet y World Wide Web, tecnología de objetos y los principales lenguajes de programación. Los fundadores de Deitel & Associates, Inc. son el Dr. Harvey M. Deitel y Paul J. Deitel. Sus clientes incluyen muchas de las empresas de cómputo más grandes del mundo, agencias gubernamentales, sectores del ejército y empresas comerciales. A lo largo de su sociedad editorial con Prentice Hall, Deitel & Associates Inc. ha publicado libros de texto de vanguardia sobre programación, libros profesionales, multimedia interactiva en CD como los *Cyber Classrooms*, cursos vía satélite y cursos de capacitación basados en Web. Deitel & Associates, Inc. y los autores pueden ser contactados mediante correo electrónico en:

`deitel@deitel.com`

Para conocer más acerca de Deitel & Associates, Inc., sus publicaciones y su currículo corporativo mundial en línea, vea las últimas páginas de este libro o visite:

`www.deitel.com`

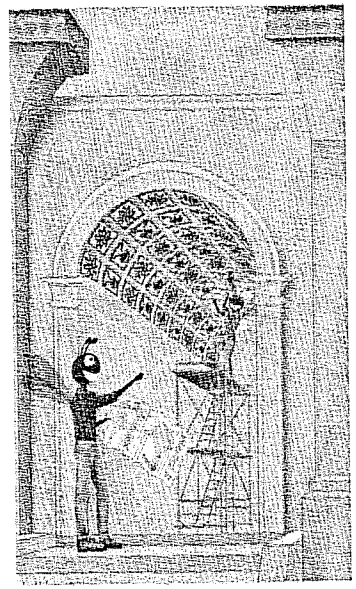
y suscríbase al boletín gratuito de correo electrónico, *DEITEL BUZZ ONLINE*, en:

`www.deitel.com/newsletter/subscribe.html`

http://librosysolucionarios.net

1

Introducción a las computadoras, Internet y Web



Objetivos

- Comprender los conceptos básicos de las ciencias computacionales.
- Familiarizarse con los distintos tipos de lenguajes de programación.
- Comprender un típico entorno de desarrollo en Java.
- Entender el papel de Java en el desarrollo de aplicaciones cliente/servidor distribuidas para Internet y Web.
- Presentar el diseño orientado a objetos con UML y los patrones de diseño.

Nuestra vida es malgastada por el detalle... simplificar, simplificar.
Henry Thoreau

Las grandes ideas requieren un gran lenguaje.
Aristófanes

La principal cualidad del lenguaje es la claridad.
Galen

Mi objeto sublime
Lo deberé llevar a cabo a tiempo.
W. S. Gilbert

Tenía un maravilloso talento para empacar estrechamente el pensamiento, haciéndolo portable.
Thomas Babington Macaulay

¡Caray, creo que de los dos, el intérprete es el más difícil de entender!
Richard Brinsley Sheridan

http://librosysolucionarios.net

Plan general

1.1 Introducción
1.2 ¿Qué es una computadora?
1.3 Organización de las computadoras
1.4 Evolución de los sistemas operativos
1.5 Computación personal, distribuida y cliente/servidor
1.6 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel
1.7 Historia de C++
1.8 Historia de Java
1.9 Bibliotecas de clases de Java
1.10 FORTRAN, COBOL, Pascal y Ada
1.11 BASIC, Visual Basic, Visual C++, C# y .NET
1.12 Internet y World Wide Web
1.13 Fundamentos de un entorno típico en Java
1.14 Generalidades acerca de Java y este libro
1.15 Acerca de los objetos: Introducción a la tecnología de objetos y el Lenguaje Unificado de Modelado
1.16 Descubrimiento de patrones de diseño: Introducción

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

1.1 Introducción

¡Bienvenido a Java! Hemos trabajado duro para crear lo que creemos será una experiencia de aprendizaje informativa, divertida y retadora para usted. Java es un poderoso lenguaje de programación de computadoras, divertido para los principiantes y apropiado para los programadores experimentados que desarrollan sistemas de información de tamaño considerable. *Cómo programar en Java, quinta edición* es un libro diseñado como una herramienta efectiva de aprendizaje para cada una de esas audiencias.

¿Cómo puede un libro ser atractivo para ambos grupos? La respuesta es que la parte central del libro se enfoca en la claridad de los programas, a través de las técnicas comprobadas de la *programación orientada a objetos*. Los principiantes aprenderán programación de la manera correcta, desde el principio. Hemos tratado de escribir en una manera clara y directa. Este libro contiene muchas ilustraciones. Tal vez lo más importante es que presenta cientos de programas funcionales en Java, y que muestra la salida que se obtiene al ejecutar estos programas en una computadora. Enseñamos las características de Java en un contexto de programas completos y funcionales en Java. A esto le llamamos el *método de código activo (LIVE-CODE™)*. Estos ejemplos están disponibles en el CD que acompaña a este libro; en el sitio Web `www.deitel.com`; y también están disponibles en nuestro producto de CD interactivo, *Java 2 Multimedia CyberClassroom, 5a. edición*.

Los primeros capítulos presentan los fundamentos de las computadoras, la programación de éstas y el lenguaje de programación Java. El material en estos capítulos muestra una base sólida para un análisis más detallado de Java en los capítulos posteriores. Los programadores experimentados tienden a leer los primeros capítulos rápidamente, y descubren que el análisis de Java en los capítulos posteriores es riguroso y retador.

La mayoría de las personas están familiarizadas con las emocionantes tareas que realizan las computadoras. Por medio de este libro de texto, usted aprenderá a programar las computadoras para que realicen esas tareas. El *software* (las instrucciones que usted escribe para indicar a la computadora que realice *acciones* y tome *decisiones*) es quien controla a las computadoras (conocidas comúnmente como *hardware*). Java, desarrollado por Sun Microsystems, es uno de los lenguajes para desarrollo de software más populares en la actualidad.

Este libro se basa en la plataforma *Java 2, Standard Edition*, la cual describe al lenguaje Java, sus bibliotecas y herramientas. Otros distribuidores pueden implementar *kits de desarrollo con Java* basándose en la Plataforma Java 2. Sun ofrece una implementación de la *Plataforma Java 2, Standard Edition* conocida como el *Kit de desarrollo de software para Java 2, Standard Edition (J2SDK)*, que incluye el conjunto mínimo de herramientas que se necesitan para escribir software en Java. Nosotros utilizamos el J2SDK versión 1.4.1, que se incluye en el CD complementario, para implementar y probar los programas de este libro. Las posteriores actualizaciones al J2SDK pueden descargarse de la página `java.sun.com/j2se`.[1]

El uso de las computadoras se está incrementando en casi cualquier campo de trabajo. En esta era donde los precios aumentan constantemente, es de sorprender que con los precios de las computadoras ocurra lo contrario debido principalmente al rápido desarrollo en la tecnología de hardware y software. Las computadoras que ocupaban grandes habitaciones y que costaban millones de dólares, hace dos décadas, ahora pueden colocarse en las superficies de chips de silicio más pequeños que una uña, y con un costo de quizá unos cuantos dólares cada uno. Irónicamente, el silicio es uno de los materiales más abundantes en el planeta (es uno de los ingredientes de la tierra común). La tecnología de los chips de silicio ha vuelto tan económica a la tecnología de la computación que cientos de millones de computadoras de uso general se encuentran actualmente ayudando a la gente de todo el mundo en empresas, en la industria, en el gobierno y en sus vidas. Dicho número podría duplicarse fácilmente en unos cuantos años.

A través de los años, muchos programadores aprendieron la metodología de programación conocida como *programación estructurada*. Usted aprenderá tanto la programación estructurada como la novedosa y excitante metodología de la *programación orientada a objetos*. ¿Por qué enseñamos ambas? La programación orientada a objetos es la metodología clave de programación utilizada hoy en día por los programadores. Usted creará y trabajará con muchos *objetos* en este curso. Sin embargo, descubrirá que la estructura interna de estos objetos se construye, a menudo, utilizando técnicas de programación estructurada. Además, la lógica requerida para manipular objetos se expresa algunas veces mediante la programación estructurada.

Java se ha convertido en el lenguaje de elección para implementar aplicaciones basadas en Internet e intranets, y para desarrollar software para dispositivos que se comunican a través de una red. ¡No se sorprenda si su nuevo equipo de sonido y otros dispositivos en su hogar pueden conectarse entre sí mediante el uso de tecnología Java! Tampoco se sorprenda cuando sus dispositivos inalámbricos, como los teléfonos celulares, radiolocalizadores y asistentes personales digitales (PDAs) se comuniquen a través de la denominada Internet inalámbrica, a través del tipo de aplicaciones de red basadas en Java que usted aprenderá a desarrollar en este libro y en el libro complementario, *Advanced Java 2 Platform How to Program*. Java ha evolucionado rápidamente en el ámbito de las aplicaciones de gran escala. Ya no es un lenguaje que se utiliza simplemente para hacer que las páginas de la World Wide Web "cobren vida". Java se ha convertido en el lenguaje preferido para satisfacer las necesidades de programación de muchas organizaciones.

Java sigue evolucionando tan rápidamente que escribimos esta quinta edición de *Cómo programar en Java* justamente seis años después de publicar la primera edición. Esta edición se basa en la *Plataforma Java 2, Standard Edition (J2SE)*. Java se ha desarrollado tan rápidamente durante los últimos años, que ahora cuenta con dos ediciones más. La *Plataforma Java 2, Enterprise Edition (J2EE)* está orientada hacia el desarrollo de aplicaciones de red distribuidas, de gran escala, y aplicaciones basadas en Web. La *Plataforma Java 2, Micro Edition (J2ME)* está orientada hacia el desarrollo de aplicaciones para dispositivos pequeños, con memoria limitada, como los teléfonos celulares, radiolocalizadores y PDAs. El número de temas a cubrir en Java se ha vuelto demasiado extenso para un solo libro. Es por esta razón que publicamos la obra *Advanced Java 2 Platform How to Program*, que se enfoca en el desarrollo de aplicaciones con la J2EE, además de que ofrece la cobertura de varios temas de alto rendimiento relacionados con la J2SE. Este libro también incluye material importante sobre la J2ME y el desarrollo de aplicaciones inalámbricas.

Está a punto de comenzar una ruta de desafíos y recompensas. Mientras tanto, si desea comunicarse con nosotros, envíenos un correo a

`deitel@deitel.com`

http://librosysolucionarios.net

1. Sun revisa el J2SDK con regularidad. Visite su sitio Web para ver si hay actualizaciones.

o explore nuestro sitio Web en

```
www.deitel.com
```

Le responderemos con prontitud. Para mantenerse al tanto de los desarrollos con Java en Deitel & Associates, regístrese para recibir nuestro boletín de correo electrónico, *The Deitel Buzz Online* en

```
www.deitel.com/newsletter/subscribe.html
```

Esperamos que disfrute la lectura de este libro tanto como nosotros disfrutamos escribirlo.

1.2 ¿Qué es una computadora?

Una *computadora* es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de millones (incluso de miles de millones) de veces más rápidas que los humanos. Por ejemplo, muchas de las computadoras personales actuales pueden realizar miles de millones de sumas por segundo. Una persona con una calculadora podría requerir toda una vida para completar el mismo número de operaciones que una computadora personal realiza en un segundo. (Puntos a considerar: ¿cómo sabría que la persona realizó los cálculos de manera correcta?, ¿cómo sabría que la computadora sumó los números de manera correcta?) ¡Las *supercomputadoras* actuales más rápidas pueden realizar miles de millones de sumas por segundo! ¡Y en los laboratorios de investigación se encuentran otras que pueden realizar billones de instrucciones en un segundo!

Las computadoras procesan los *datos* bajo el control de conjuntos de instrucciones llamadas *programas de cómputo*. Estos programas de cómputo guían a la computadora a través de conjuntos ordenados de acciones especificadas por gente llamada *programadores de computadoras*.

Una computadora está compuesta por varios dispositivos (como el teclado, monitor, "ratón", discos, memoria, DVD, CD-ROM y unidades de procesamiento) conocidos como *hardware*. A los programas de cómputo que se ejecutan en una computadora se les denomina *software*. Los costos de las piezas de hardware han disminuido de manera espectacular en años recientes, al punto en el que las computadoras personales se han convertido en artículos domésticos. Desafortunadamente, los costos para el desarrollo de programas se incrementan de manera constante conforme los programadores desarrollan aplicaciones más complejas y poderosas, sin que exista una mejora significativa en la tecnología para el desarrollo de software. En este libro aprenderá métodos comprobados para el desarrollo de software que pueden reducir sus costos (programación estructurada, mejoramiento paso a paso, uso de funciones, programación basada en objetos, programación orientada a objetos y diseño orientado a objetos).

1.3 Organización de las computadoras

Independientemente de la apariencia física, casi siempre las computadoras pueden representarse mediante seis *unidades* o secciones *lógicas*. Éstas son:

1. *Unidad de entrada*. Es la sección "receptora" de la computadora. Obtiene información (datos y programas de cómputo) desde varios dispositivos de entrada y pone esta información a disposición de las otras unidades para que pueda procesarse. La mayor parte de la información se introduce a través de los teclados y ratones. En la actualidad, la información también puede introducirse hablando con su computadora, digitalizando imágenes y mediante la recepción de información desde una red, como Internet.
2. *Unidad de salida*. Ésta es la sección de "embarque" de la computadora. Toma información que ya ha sido procesada por la computadora y la coloca en los diferentes *dispositivos de salida*, para que esté disponible fuera de la computadora. Hoy en día, la mayor parte de la información de salida de las computadoras se despliega en el monitor, se imprime en papel o se utiliza para controlar otros dispositivos. Las computadoras también pueden dar salida a su información a través de redes como Internet.
3. *Unidad de memoria*. Esta sección funciona en la computadora como un "almacén" de acceso rápido, pero con relativa baja capacidad. Retiene la información que se introduce a través de la unidad de entrada, para que la información pueda estar disponible de manera inmediata para procesarla cuando sea necesario. La unidad de memoria también retiene la información procesada hasta que ésta pueda ser colocada en los dispositivos de salida por la unidad de salida. Con frecuencia, a la unidad de memoria se le llama *memoria* o *memoria primaria*.
4. *Unidad aritmética y lógica (ALU)*. Ésta es la sección de "manufactura" de la computadora. Es la responsable de realizar cálculos como suma, resta, multiplicación y división. Contiene los mecanismos de decisión que permiten a la computadora hacer cosas como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no.
5. *Unidad central de procesamiento (CPU)*. Ésta es la sección "administrativa" de la computadora. Es quien coordina y supervisa la operación de las demás secciones. La CPU le indica a la unidad de entrada cuándo debe grabarse la información dentro de la unidad de memoria, a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y a la unidad de salida cuándo enviar la información desde la unidad de memoria hasta ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples unidades de procesamiento y, por lo tanto, pueden realizar múltiples operaciones de manera simultánea (a estas computadoras se les conoce como *multiprocesadoras*).
6. *Unidad de almacenamiento secundario*. Éste es el "almacén" de alta capacidad y de larga duración de la computadora. Los programas o datos que no se encuentran en ejecución por las otras unidades, normalmente se colocan en dispositivos de almacenamiento secundario (como discos) hasta que son requeridos de nuevo, posiblemente horas, días, meses o incluso años después. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el necesario para acceder a la de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria principal.

1.4 Evolución de los sistemas operativos

Las primeras computadoras eran capaces de realizar solamente una *tarea* o *trabajo* a la vez. A esta forma de operación de la computadora a menudo se le conoce como *procesamiento por lotes* (batch) de un solo usuario. La computadora ejecuta un solo programa a la vez, mientras procesa los datos en grupos o *lotes*. En estos primeros sistemas, los usuarios generalmente asignaban sus trabajos a un centro de cómputo que lo introducía en paquetes de tarjetas perforadas. A menudo tenían que esperar horas, o incluso días, antes de que sus resultados impresos regresaran a sus escritorios.

Los sistemas de software denominados *sistemas operativos* se desarrollaron para facilitar el uso de la computadora. Los primeros sistemas operativos administraban la suave transición entre trabajos. Esto minimizó el tiempo necesario para que los operadores de computadoras pasaran de una tarea a otra, y por consiguiente incrementó la cantidad de trabajo, o el *flujo de datos*, que las computadoras podían procesar.

Conforme las computadoras se volvieron más poderosas, se hizo evidente que un proceso por lotes para un solo usuario rara vez aprovechaba todos los recursos de la computadora de manera eficiente, debido al tiempo que se malgastaba esperando a que los lentos dispositivos de entrada/salida completaran sus tareas. Se pensó que era posible realizar muchos trabajos o tareas que podrían *compartir* los recursos de la computadora y lograr un uso más eficiente de ésta. A esto se le conoce como *multiprogramación*. La multiprogramación significa la operación "simultánea" de muchas tareas dentro de la computadora (ésta comparte sus recursos entre los trabajos que compiten por su atención).

En la década de los sesenta, varios grupos en la industria y en las universidades marcaron la pauta de los sistemas operativos de *tiempo compartido*. El tiempo compartido es un caso especial de la multiprogramación, en el cual los usuarios acceden a la computadora a través de *terminales*, que por lo general son dispositivos compuestos por un teclado y un monitor. En un típico sistema de cómputo de tiempo compartido puede haber docenas o incluso cientos de usuarios compartiendo la computadora al mismo tiempo. La computadora en realidad no ejecuta los procesos de todos los usuarios a la vez. Lo que hace es ejecutar una pequeña porción del trabajo de un usuario y después procede a dar servicio al siguiente usuario. La computadora hace el trabajo tan rápidamente que puede proporcionar el servicio a cada usuario varias veces por segundo. Así, los programas de los usuarios *aparentemente* se ejecutan de manera simultánea. Una ventaja del tiempo compartido es que el usuario recibe respuestas casi inmediatas a las peticiones, en vez de tener que esperar los resultados durante largos periodos, como en los comienzos de la computación.

http://librosysolucionarios.net

1.5 Computación personal, distribuida y cliente/servidor

En 1977, Apple Computer popularizó el fenómeno de la *computación personal*. Al principio era el sueño de todo aficionado. Las computadoras se hicieron lo suficientemente económicas para que la gente las pudiera adquirir para su uso personal o para negocios. En 1981, IBM, el vendedor de computadoras más grande del mundo, introdujo la PC de IBM. Literalmente, de la noche a la mañana, la computación personal se posicionó en las empresas, en la industria y en las instituciones gubernamentales.

Sin embargo, estas computadoras eran unidades "independientes" (la gente hacía el trabajo en su propia máquina y transportaba sus discos de un lado a otro para compartir información; a esto se le conoce comúnmente como "sneakernet"). Aunque las primeras computadoras personales no eran lo suficientemente poderosas para compartir el tiempo entre muchos usuarios, estas máquinas podían interconectarse mediante redes,algunas veces a través de líneas telefónicas y otras mediante *redes de área local (LANs)* dentro de la empresa. Esto derivó en el fenómeno denominado *computación distribuida*, en donde la computación de una empresa, en vez de realizarse estrictamente dentro de un centro de cómputo, se distribuye mediante redes a los sitios en donde se realiza el trabajo de la empresa. Las computadoras personales eran lo suficientemente poderosas para manejar los requerimientos de cómputo de usuarios individuales, y para manejar las tareas básicas de comunicación que involucraban la transferencia de información entre una computadora y otra, de manera electrónica.

Las computadoras personales actuales son tan poderosas como las máquinas de un millón de dólares de hace apenas una década. Las máquinas de escritorio más poderosas (denominadas *estaciones de trabajo*) proporcionan al usuario enormes capacidades. La información se comparte fácilmente a través de redes de computadoras, en donde algunas computadoras denominadas *servidores de archivos* ofrecen un almacén de datos común que puede ser utilizado por computadoras *cliente* distribuidas en toda la red, de ahí el término de computación *cliente/servidor*. Java se está utilizando ampliamente para escribir software para redes de computadoras y para aplicaciones cliente/servidor distribuidas. Los sistemas operativos actuales más populares como UNIX, Linux, Mac OS X y Windows proporcionan el tipo de capacidades que explicamos en esta sección.

1.6 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de los cuales los comprende directamente la computadora, mientras que otros requieren pasos intermedios de *traducción*. En la actualidad se utilizan cientos de lenguajes de computación; los cuales se dividen en tres tipos generales:

1. Lenguajes máquina.
2. Lenguajes ensambladores.
3. Lenguajes de alto nivel.

Cualquier computadora puede entender de manera directa sólo su propio *lenguaje máquina*. El lenguaje máquina es el "lenguaje natural" de una computadora en particular y está definido por el diseño del hardware de dicha computadora. Por lo general, los lenguajes máquina consisten en cadenas de números [que finalmente se reducen a unos (1) y ceros (0)] que instruyen a las computadoras para realizar sus operaciones más elementales, una por una. Los lenguajes máquina son *dependientes de la máquina* (es decir, un lenguaje máquina en particular puede usarse solamente en un tipo de computadora). Dichos lenguajes son difíciles de comprender para los humanos, como puede ver en el programa de lenguaje máquina que se presenta en la siguiente sección, el cual suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
+1300042774
+1400593419
+1200274027
```

La programación en lenguaje máquina era demasiado lenta y tediosa para la mayoría de los programadores. En vez de utilizar las cadenas de números que las computadoras podían entender directamente, los programadores empezaron a utilizar abreviaturas del inglés para representar las operaciones elementales. Estas abreviaturas formaron la base de los *lenguajes ensambladores*. Los *programas traductores* conocidos como *ensambladores* se desarrollaron para convertir los programas en lenguaje ensamblador a lenguaje máquina, a la velocidad de

la computadora. La siguiente sección muestra un programa en lenguaje ensamblador que también suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto, pero es mucho más claro que su equivalente en lenguaje máquina:

```
LOAD    SUELDOBASE
ADD     SUELDOEXTRA
STORE   SUELDOBRUTO
```

Aunque este código es más claro para los humanos, las computadoras no lo pueden entender sino hasta que se traduce en lenguaje máquina.

El uso de las computadoras se incrementó rápidamente con la llegada de los lenguajes ensambladores, pero la programación en estos lenguajes aún requería de muchas instrucciones para llevar a cabo incluso hasta las tareas más simples. Para agilizar el proceso de programación se desarrollaron los *lenguajes de alto nivel*, en donde podían escribirse instrucciones individuales para realizar tareas importantes. Los programas traductores, denominados compiladores, convierten programas en lenguaje de alto nivel a lenguaje máquina. Los lenguajes de alto nivel permiten a los programadores escribir instrucciones que son muy similares al inglés común, y contienen la notación matemática común. Un programa de nómina escrito en un lenguaje de alto nivel podría contener una instrucción como la siguiente:

```
sueldoBruto = sueldoBase + sueldoExtra
```

Obviamente, los niveles de alto nivel son mucho más recomendables, desde el punto de vista del programador, que los lenguajes máquina o ensamblador. C, C++ y Java son los lenguajes de programación de alto nivel más poderosos y más ampliamente utilizados.

El proceso de compilación de un programa escrito en lenguaje de alto nivel a un lenguaje máquina puede tardar un tiempo considerable. Los programas *intérpretes* se desarrollaron para ejecutar programas en lenguaje de alto nivel directamente, sin necesidad de compilarlos en lenguaje máquina. Aunque los programas compilados se ejecutan mucho más rápido que los programas interpretados, los intérpretes son populares dentro de entornos de desarrollo de programas, en los cuales los programas se recompilan frecuentemente, a medida que se agregan nuevas características y se corrigen los errores. Una vez que se desarrolla un programa, se puede producir una versión compilada para ejecutarse con la mayor eficiencia.

1.7 Historia de C++

C++ evolucionó de C, el cual a su vez evolucionó de dos lenguajes de programación anteriores, BCPL y B. En 1967, Martin Richards desarrolló BCPL como un lenguaje para escribir software para sistemas operativos y compiladores. Ken Thompson modeló muchas características en su lenguaje B a partir del trabajo de sus contrapartes en BCPL, y utilizó a B para crear las primeras versiones del sistema operativo UNIX, en los laboratorios Bell en 1970.

El lenguaje C evolucionó a partir de B, gracias al trabajo de Dennis Ritchie en los laboratorios Bell, y se implementó originalmente en 1972. Inicialmente, C se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX. En la actualidad, todos los nuevos sistemas operativos importantes se escriben en C o C++.

La amplia utilización de C para distintos tipos de computadoras (en ocasiones llamadas *plataformas de hardware*) condujo a muchas variaciones. Éstas eran similares, pero a menudo incompatibles. Esto se convirtió en un serio problema para los programadores que necesitaban escribir programas portables, que pudieran ejecutarse en varias plataformas. Entonces, se hizo evidente que se necesitaba una versión estándar de C. En 1983, se creó el comité técnico X3J11 bajo la supervisión del Comité nacional estadounidense de estándares en computadoras y procesamiento de información (X3) para "proporcionar una definición del lenguaje clara e independiente de la máquina". En 1989 se aprobó el estándar. El Instituto nacional estadounidense de estándares (ANSI) cooperó con la International Standard Organization (ISO), para estandarizar C a nivel mundial; el documento conjunto del estándar se publicó en 1990 y se conoce como ANSI/ISO 9899: 1990. La segunda edición de Kernighan y Ritchie,[2] publicada en 1988, menciona esta versión denominada ANSI C, una versión del lenguaje que aún se utiliza a nivel mundial.

http://librosysolucionarios.net

2. Kernighan, B. W. y D. M. Ritchie. *The C Programming Language: Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1988.

A principios de la década de los ochenta, Bjarne Stroustrup desarrolló una extensión de C en los laboratorios Bell: C++. Este lenguaje proporciona un conjunto de características que "pulen" al lenguaje C pero, lo más importante es que proporciona la capacidad de una *programación orientada a objetos* (que describiremos con más detalle en la sección 1.15). C++ también se estandarizó mediante los comités ANSI e ISO.

Una revolución se está gestando en la comunidad del software. Escribir software rápida, correcta y económicamente es aún una meta escurridiza, en una época en que la demanda de nuevo y más poderoso software se encuentra a la alza. Los *objetos* son, esencialmente, componentes reutilizables de software que modelan piezas de programas de software en términos de *propiedades* (como el nombre, color y tamaño) y *comportamientos* (como calcular, desplazarse y comunicarse). Los desarrolladores de software están descubriendo que el uso de una metodología de diseño e implementación modular y orientada a objetos puede hacer más productivos a los grupos de desarrollo de software, que mediante las populares técnicas de programación anteriores, como la programación estructurada. Los programas orientados a objetos son, a menudo, más fáciles de entender, corregir y modificar.

Se han desarrollado muchos otros lenguajes orientados a objetos, incluyendo a Smalltalk, desarrollado en el Centro de investigaciones de Palo Alto (PARC) de Xerox. Smalltalk es un lenguaje orientado a objetos puros (literalmente todo es un objeto). C++ es un lenguaje híbrido (es posible programar tanto en estilo C, como en estilo orientado a objetos, o en ambos).

1.8 Historia de Java

Tal vez la contribución más importante a la fecha, por parte de la revolución del microprocesador, es que hizo posible el desarrollo de las computadoras personales, que ahora suman cientos de millones a nivel mundial. Las computadoras personales han tenido un profundo impacto en la vida de las personas, y en la manera en que las empresas realizan y administran su negocio.

Muchas personas creen que la siguiente área importante en la que los microprocesadores tendrán un profundo impacto es en los dispositivos electrónicos para uso doméstico. Al reconocer esto, Sun Microsystems patrocinó en 1991 un proyecto interno de investigación denominado Green. El proyecto desembocó en el desarrollo de un lenguaje basado en C++ al que su creador, James Gosling, llamó Oak debido a un roble que tenía a la vista desde su ventana en las oficinas de Sun. Posteriormente se descubrió que ya existía un lenguaje de programación con el mismo nombre. Cuando un grupo de gente de Sun visitó una cafetería local, sugirieron el nombre *Java* (una variedad de café) y así se quedó.

Pero el proyecto Green tuvo algunas dificultades. El mercado para los dispositivos electrónicos inteligentes de uso doméstico no se desarrollaba tan rápido como Sun había anticipado. Peor aún, un contrato importante por el que Sun había competido se le otorgó a otra empresa. De manera que el proyecto corría el riesgo de cancelarse. Pero para su buena fortuna, la popularidad de World Wide Web explotó en 1993 y la gente de Sun se dio cuenta inmediatamente del potencial de Java para agregar *contenido dinámico* y animaciones a las páginas Web. Esto trajo nueva vida al proyecto.

Sun anunció formalmente a Java en una conferencia importante que tuvo lugar en mayo de 1995. Por lo general, un evento como éste no habría generado mucha atención. Sin embargo, Java generó un interés inmediato en la comunidad de negocios, debido al fenomenal interés en World Wide Web. En la actualidad, Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores de World Wide Web (las computadoras que proporcionan el contenido que vemos en nuestros navegadores Web), para proporcionar aplicaciones para los dispositivos domésticos (como teléfonos celulares, radiolocalizadores y asistentes digitales personales) y para muchos otros propósitos.

1.9 Bibliotecas de clases de Java

Los programas en Java constan de varias piezas llamadas *clases*. Estas clases incluyen piezas llamadas *métodos*, los cuales realizan tareas y devuelven información cuando completan esas tareas. Usted puede crear cada una de las piezas que necesita para formar un programa en Java. Sin embargo, la mayoría de los programadores en Java aprovechan las ricas colecciones de clases existentes en las *bibliotecas de clases de Java*, que también se conocen como *APIs (Interfaces de programación de aplicaciones) de Java*. Por lo tanto, en realidad existen dos fundamentos para conocer el "mundo" de Java. El primero es el lenguaje Java en sí, de manera que

usted pueda programar sus propias clases; el segundo son las clases incluidas en las extensas bibliotecas de clases de Java. A lo largo de este libro hablaremos sobre muchas bibliotecas de clases. Las bibliotecas de clases las proporcionan principalmente los vendedores de compiladores, pero muchas de ellas las proporcionan vendedores de software independientes (ISVs).

Observación de ingeniería de software 1.1

Utilice un método de construcción en bloques para crear programas. Evite reinventar la rueda; use piezas existentes. Esta práctica, conocida como reutilización de software *es vital para la programación orientada a objetos.*

[*Nota:* Incluimos muchas de estas *Observaciones de ingeniería de software* a lo largo de este texto para explicar los conceptos que afectan y mejoran la arquitectura y calidad de los sistemas de software en general y, en particular, de sistemas de software de gran escala. También resaltamos las *Buenas prácticas de programación* (que le ayudarán a escribir programas más claros, comprensibles, de fácil mantenimiento, y fáciles de probar y *corregir*, o eliminar errores de programación), los *Errores comunes de programación* (problemas de los que se tiene que cuidar, de manera que no los cometa dentro de sus programas), *Tips de rendimiento* (técnicas que le ayudarán a escribir programas que se ejecuten más rápido y ocupen menos memoria), *Tips de portabilidad* (técnicas que le ayudarán a escribir programas que se ejecuten, con poca o ninguna modificación, en una variedad de computadoras; estos tips también incluyen observaciones generales acerca de cómo logra Java su alto grado de portabilidad), *Tips para prevenir errores* (técnicas que le ayudarán a eliminar errores de sus programas y, lo que es más importante, técnicas que le ayudarán a escribir programas libres de errores) y *Observaciones de apariencia visual* (técnicas que le ayudarán a diseñar la apariencia visual de sus interfaces gráficas de usuario, además de facilitar su uso). Muchas de estas técnicas y prácticas son sólo guías; usted deberá, sin duda, desarrollar su propio estilo de programación.]

Observación de ingeniería de software 1.2

Cuando se programa en Java, generalmente se utilizan los siguientes bloques de construcción: clases de las bibliotecas de clases, clases y métodos creadas por uno mismo, y clases y métodos creados por otros y puestos a disponibilidad suya.

La ventaja de crear sus propias clases y métodos es que sabe exactamente cómo funcionan y puede examinar el código en Java. La desventaja es que toman mucho tiempo y requiere de un gran esfuerzo diseñar y desarrollar nuevas clases y métodos.

Tip de rendimiento 1.1

Utilizar las clases y métodos de las APIs de Java en vez de escribir sus propias versiones puede mejorar el rendimiento de sus programas, ya que estas clases y métodos están escritos cuidadosamente para funcionar de manera eficiente y correcta. Esta técnica también mejora la velocidad de prototipo en el desarrollo del programa (es decir, el tiempo requerido para desarrollar un nuevo programa y hacer que su primera versión pueda ejecutarse).

Tip de portabilidad 1.1

Utilizar las clases y métodos de las APIs de Java en vez de escribir sus propias versiones mejora la portabilidad de sus programas, ya que estas clases y métodos se incluyen en todas las implementaciones de Java.

Observación de ingeniería de software 1.3

Existen diversas bibliotecas de clases que contienen componentes reutilizables de software, las cuales están disponibles a través de Internet y la Web. Muchas de estas bibliotecas están disponibles en forma gratuita.

1.10 FORTRAN, COBOL, Pascal y Ada

Se han desarrollado cientos de lenguajes de alto nivel, pero sólo unos cuantos han logrado una amplia aceptación. *FORTRAN* (FORmula TRANslator) fue desarrollado por IBM Corporation en la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. FORTRAN se utiliza ampliamente todavía, en especial en aplicaciones de ingeniería.

COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado en 1959 por fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria. COBOL se utiliza en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos. Mediante algunas estimaciones, más de la mitad de todo el software de negocios aún se programa en COBOL.

http://librosysolucionarios.net

Durante la década de los sesentas, muchos de los grandes esfuerzos para el desarrollo de software encontraron severas dificultades. Los itinerarios de software generalmente se retrasaban, los costos rebasaban en gran medida a los presupuestos y los productos terminados no eran confiables. La gente comenzó a darse cuenta de que el desarrollo de software era una actividad mucho más compleja de lo que habían imaginado. Las actividades de investigación en la década de sesentas dieron como resultado la evolución de la *programación estructurada* (un método disciplinado para escribir programas que sean más claros, fáciles de probar y corregir, y más fáciles de modificar que los programas no estructurados).

Uno de los resultados más tangibles de esta investigación fue el desarrollo del lenguaje de programación Pascal por el profesor Niklaus Wirth, en 1971. Pascal, cuyo nombre se debe al matemático y filósofo Blaise Pascal del siglo diecisiete, se diseño para la enseñanza de la programación estructurada en ambientes académicos, y de inmediato se convirtió en el lenguaje de programación preferido en la mayoría de las universidades. Desafortunadamente, el lenguaje carece de muchas de las características necesarias para poder utilizarse en aplicaciones comerciales, industriales y gubernamentales, por lo que no ha sido muy aceptado fuera de las universidades.

El lenguaje de programación *Ada* se desarrolló bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DOD) durante la década de los setenta y los primeros años de la década de los ochenta. Cientos de lenguajes independientes se utilizaron para producir los sistemas de software masivos de comando y control del departamento de defensa. Éste quería un solo lenguaje que pudiera satisfacer la mayoría de sus necesidades. El nombre del lenguaje es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A Lady Lovelace se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de 1800 (para la Máquina Analítica, un dispositivo de cómputo mecánico diseñado por Charles Babbage). Una de las características importantes de Ada se conoce como *multitarea*, la cual permite a los programadores especificar que muchas actividades ocurrirán en paralelo. Java, a través de una técnica que explicaremos y que se conoce como *subprocesamiento múltiple*, también permite a los programadores escribir programas con actividades paralelas.

1.11 BASIC, Visual Basic, Visual C++, C# y .NET

El lenguaje de programación BASIC (Beginner's All-Purpose Symbolic Instruction Code, Código de instrucciones simbólicas de uso general para principiantes) fue desarrollado a mediados de la década de los sesenta por los profesores John Kemeny y Thomas Kurtz, del Dartmouth College, como un lenguaje para escribir programas simples. El propósito principal de BASIC era que los principiantes se familiarizaran con las técnicas de programación. Visual Basic se introdujo en 1991 para simplificar el desarrollo de aplicaciones para Microsoft Windows.

Visual Basic .NET está diseñado para la nueva plataforma de programación de Microsoft: .NET. Las primeras versiones de Visual Basic contaban con características orientadas a objetos, pero Visual Basic .NET ofrece una orientación a objetos mejorada, además de que hace uso de la poderosa biblioteca de .NET conocida como Biblioteca de clases del Framework (FCL), en la que se incluyen componentes de software reutilizables.

Visual C++ es una implementación de Microsoft de C++, que incluye extensiones propias de Microsoft para el lenguaje. En un principio, la programación de gráficos y GUI con Visual C++ se implementaba mediante la MFC (Microsoft Foundation Classes). Ahora, con la introducción de .NET, Microsoft proporciona una biblioteca común (FCL) para implementar GUIs, gráficos, redes, subprocesamiento múltiple y otras características. Esta biblioteca se comparte entre Visual Basic, Visual C++, C# (el nuevo lenguaje de Microsoft) y muchos otros lenguajes que Microsoft y otros distribuidores están poniendo a disposición de .NET.

El avance en las herramientas de programación (por ejemplo, C++ y Java) y en los dispositivos electrodomésticos (por ejemplo, teléfonos celulares y PDAs) creó problemas y nuevos requerimientos. La integración de componentes de software de varios lenguajes resultó difícil, y los problemas de instalación eran comunes debido a que las nuevas versiones de los componentes compartidos eran incompatibles con el software anterior. Los desarrolladores descubrieron también que necesitaban aplicaciones para Web, a las que pudieran acceder y utilizar mediante Internet. Como resultado de la popularidad de los dispositivos electrónicos móviles, los desarrolladores reconocieron la necesidad de software que estuviera accesible para todos, y disponible a través de casi cualquier tipo de dispositivo. Para hacer frente a estas necesidades, Microsoft anunció su *iniciativa .NET* y el lenguaje de programación *C#*.

http://librosysolucionarios.net

De manera similar a Java, la *plataforma .NET* permite la distribución de aplicaciones basadas en Web a muchos dispositivos (incluso teléfonos celulares) y computadoras de escritorio. El lenguaje de programación *C#*, desarrollado en Microsoft por un equipo dirigido por Anders Hejlsberg y Scott Wiltamuth, fue diseñado específicamente para la plataforma .NET como un lenguaje que permitiera a los programadores migrar fácilmente a .NET.

1.12 Internet y World Wide Web

Internet (una red global de computadoras) se desarrolló hace más de cuatro décadas, y su patrocinio estuvo a cargo del Departamento de Defensa de los Estados Unidos. Diseñada en un principio para conectar los sistemas de cómputo principales de aproximadamente una docena de universidades y organizaciones de investigación, Internet es ahora utilizada por cientos de millones de computadoras en todo el mundo.

Con la introducción de *World Wide Web* (que permite a los usuarios de computadora localizar y ver documentos basados en multimedia, sobre casi cualquier tema, a través de Internet), Internet se ha convertido explosivamente en uno de los principales mecanismos de comunicación en todo el mundo.

Internet y World Wide Web se encuentran, sin duda, entre las creaciones más importantes y profundas de la humanidad. En el pasado, la mayoría de las aplicaciones de computadora se ejecutaban en equipos que no estaban conectados entre sí. Las aplicaciones de la actualidad pueden diseñarse para intercomunicarse entre los cientos de millones de computadoras en todo el mundo. Internet mezcla las tecnologías de la computación y las comunicaciones. Facilita nuestro trabajo. Hace que la información esté accesible en forma instantánea y conveniente para todo el mundo. Hace posible que los individuos y negocios pequeños locales obtengan una exposición mundial. Está cambiando la naturaleza de la manera en que se hacen los negocios. La gente puede buscar los mejores precios para casi cualquier producto o servicio. Los miembros de las comunidades con intereses especiales pueden mantenerse en contacto unos con otros. Los investigadores pueden estar inmediatamente al tanto de los últimos descubrimientos.

Cómo programar en Java, quinta edición, presenta técnicas de programación que permiten a las aplicaciones en Java utilizar Internet y Web para interactuar con otras aplicaciones. Estas técnicas, junto con las que se describen en nuestro libro *Advanced Java 2 Platform How to Program*, permiten a los programadores de Java desarrollar el tipo de aplicaciones distribuidas de nivel empresarial que se utilizan actualmente en la industria. Se pueden escribir aplicaciones en Java para ejecutarse en cualquier plataforma de computadora, lo cual produce a las empresas un considerable ahorro en tiempo de desarrollo de sistemas y costos. Si a usted le interesa desarrollar aplicaciones que se ejecuten a través de Internet y Web, aprender Java puede ser la clave para que reciba oportunidades retadoras y remuneradoras en su profesión.

1.13 Fundamentos de un entorno típico en Java

La siguiente explicación define los pasos típicos para crear y ejecutar un programa en Java, utilizando el entorno de desarrollo de Java. Estos pasos se muestran en la figura 1.1 y se explican en el siguiente texto.

Por lo general, los programas en Java pasan a través de cinco fases para poder ejecutarse (figura 1.1). Estas fases son: *edición, compilación, carga, verificación* y *ejecución*. Hablamos sobre estos conceptos en el contexto del *Kit de desarrollo de software para Java 2 (J2SDK) versión 1.4.1*, el cual se incluye en el CD que acompaña a este libro. *Siga cuidadosamente las instrucciones de instalación para el J2SDK que se proporcionan en el CD, para asegurarse de configurar su computadora apropiadamente para compilar y ejecutar programas en Java.* Encontrará las instrucciones completas de instalación para el J2SDK en el sitio Web de Java de Sun en

```
developer.java.sun.com/developer/onlineTraining/new2java/programming/learn/
```

[*Nota:* El sitio Web[3] anterior proporciona las instrucciones de instalación para UNIX/Linux, Windows y MacOS. Si usted no utiliza uno de estos sistemas operativos, consulte los manuales del entorno de Java de su sistema, o pregunte a su instructor cómo puede lograr estas tareas en su entorno.]

[3] Recuerde que los vínculos Web se interrumpen cuando las compañías hacen modificaciones para mejorar sus sitios Web. Si encuentra problemas con los vínculos en este libro, consulte la fe de erratas en nuestro sitio Web `www.deitel.com` y por favor notifíquenos por correo electrónico a `deitel@deitel.com`.

La fase 1 consiste en editar un archivo. Esto se logra mediante un *programa de edición* (conocido comúnmente como *editor*). El programador escribe un programa en Java utilizando el editor, y realiza las correcciones, si es necesario. Cuando el programador especifica que debe guardarse el archivo en el editor, el programa se guarda en un dispositivo de almacenamiento secundario, tal como un disco. Los nombres de archivo de los programas en Java terminan con la *extensión* `.java`. Una extensión es la porción de un nombre de archivo que identifica el tipo de datos que éste contiene. La extensión `.java` del nombre de archivo indica que éste contiene código fuente en el lenguaje de programación Java. Los dos editores más utilizados en sistemas UNIX/Linux son `vi` y `emacs`. En Windows bastará con usar programas simples de edición, como el Bloc de notas. Los

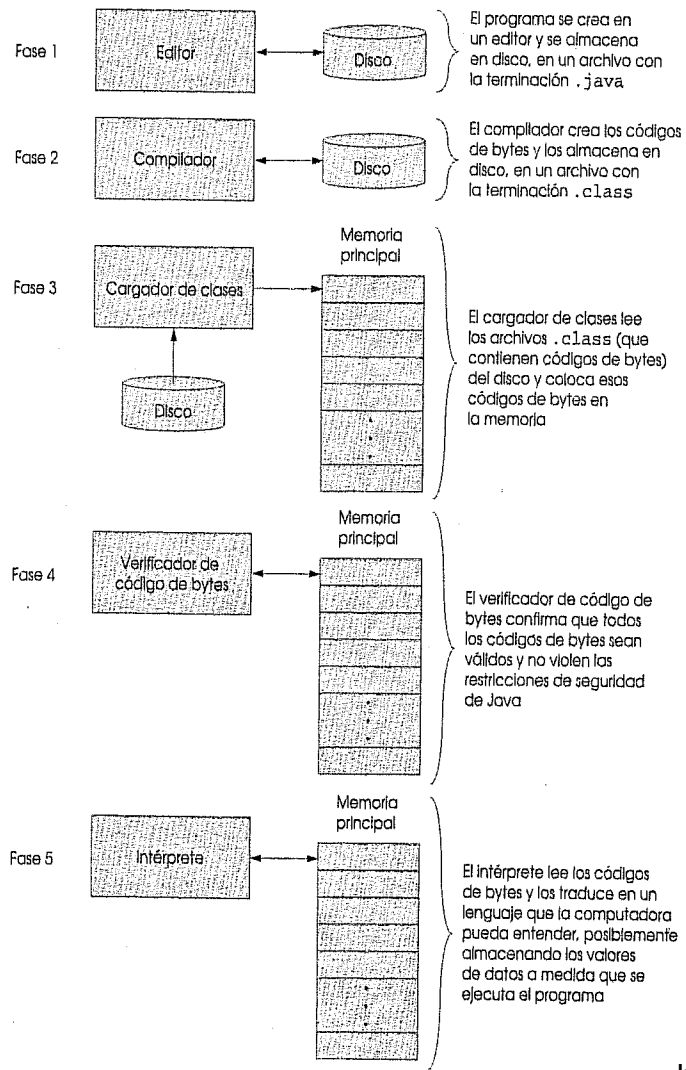


Figura 1.1 Entorno típico en Java.

http://librosysolucionarios.net

entornos integrados de desarrollo (IDEs) de Java, como Sun™ ONE Studio,[4] tienen editores de texto que están integrados al entorno de programación. En este libro asumimos que el lector sabe cómo editar un archivo.

En la fase 2 (que se describe con más de talle en los capítulos 2 y 3), el programador proporciona el comando `javac` para compilar el programa. El compilador de Java traduce el programa en Java a *códigos de bytes*: las instrucciones que el intérprete de Java puede entender. Para compilar un programa llamado `Bienvenido.java`, escriba

```
javac Bienvenido.java
```

en la ventana de comandos de su sistema (es decir, el *indicador de MS-DOS* en Windows 95/98/ME, el *Símbolo del sistema* en Windows NT/2000/XP o el *indicador de shell* en UNIX/Linux). Si el programa se compila correctamente, el compilador produce un archivo llamado `Bienvenido.class`. Este archivo contiene los códigos de bytes que se interpretarán durante la fase de ejecución.

La fase 3 se conoce como *carga*. El programa debe colocarse en memoria antes de ejecutarse. De esto se encarga el *cargador de clases*, que toma los archivos `.class` que contienen los códigos de bytes y los transfiere a la memoria principal. Después, el archivo `.class` puede cargarse desde un disco en su sistema o a través de una red (como la de su universidad local, la red de la empresa o incluso desde Internet).

A medida que se cargan las clases, el *verificador de códigos de bytes* se encarga de verificar sus códigos de bytes en la fase 4. Este proceso de verificación asegura que los códigos de bytes de las clases sean válidos y que no violen las restricciones de seguridad de Java. Java implementa una estrecha seguridad, ya que los programas en Java que llegan a través de la red no deben ser capaces de dañar sus archivos o su sistema (como podrían hacerlo los virus de computadora). Tome en cuenta que la verificación del código de bytes también se lleva a cabo en aplicaciones que descargan clases a través de una red.

Finalmente, en la fase 5, el intérprete, bajo el control del sistema operativo, interpreta el programa un código de bytes a la vez, realizando en esta forma las acciones especificadas por el programa. Existen dos tipos de programas para los cuales ocurre este proceso: *aplicaciones* y *applets*.[5] Una aplicación es un programa (como un programa de procesamiento de palabras, de hojas de cálculo, de dibujo o de correo electrónico) que generalmente se guarda y ejecuta desde el equipo local del usuario. Un applet es un pequeño programa que generalmente se guarda en un equipo remoto, al cual se conectan los usuarios mediante un navegador Web. El equipo remoto se conoce como un *servidor Web*. Los applets se cargan desde un equipo remoto en el navegador, se ejecutan en éste y se descartan cuando termina su ejecución. Para ejecutar un applet de nuevo, el usuario debe dirigir un navegador a la ubicación apropiada en World Wide Web y volver a cargar el programa en el navegador.

Las aplicaciones se cargan en memoria y se ejecutan, utilizando el *intérprete de Java* mediante el comando `java`. Al ejecutar una aplicación de Java llamada `Bienvenido`, el comando

```
java Bienvenido
```

invoca al intérprete para la aplicación `Bienvenido`, y hace que el cargador de clases cargue la información utilizada en el programa `Bienvenido`. [*Nota:* El intérprete de Java también se conoce como la *Máquina virtual de Java*, o la *JVM*.]

Los navegadores Web como *Netscape* o *Microsoft Internet Explorer* se utilizan para ver documentos en World Wide Web, los cuales se conocen como documentos *HTML (Lenguaje de marcado de hipertexto)*. HTML describe el formato de un documento de tal manera que el navegador pueda entenderlo. En la sección 3.4 daremos una introducción a HTML; para obtener un análisis detallado acerca de HTML y de otras tecnologías de programación en Internet, consulte nuestro texto *Internet and World Wide Web How to Program, Second Edition*.) Un documento en HTML puede contener un applet de Java. Cuando el navegador ve la referencia a un applet en un documento HTML, inicia el cargador de clases de Java para cargar el applet (generalmente desde la ubicación en que se almacena el documento HTML). Todos los navegadores que cuentan con soporte para

4. *Sun™ ONE Studio, Community Edition* se incluye en el CD que acompaña a este libro. *Sun ONE Studio, Community Edition* se ejecuta en casi todas las plataformas importantes. Nuestros programas de ejemplo deben operar apropiadamente con cualquier entorno integrado de desarrollo de Java que tenga soporte para la Plataforma Java 2, Standard Edition, versión 1.4.1.
5. En capítulos posteriores hablaremos sobre otro tipo de programa en Java, conocido como "servlet", el cual extiende la funcionalidad de un servidor Web.

Java tienen un intérprete de Java integrado (es decir, una JVM). Una vez que se carga el applet, el intérprete de Java lo ejecuta. Los applets también pueden ejecutarse desde la línea de comandos, mediante el *comando* `appletviewer` que se incluye con el J2SDK (el conjunto de herramientas que incluyen el compilador (`javac`), el intérprete (`java`), `appletviewer` y otras herramientas utilizadas por los programadores de Java. Al igual que Netscape y Microsoft Internet Explorer, el `appletviewer` (visor de comandos) requiere un documento HTML para poder invocar a un applet. Por ejemplo, si el archivo `Bienvenido.html` hace referencia al applet `Bienvenido`, el comando `appletviewer` se utiliza de la siguiente manera:

```
appletviewer Bienvenido.html
```

Este comando ocasiona que el cargador de clases cargue la información utilizada en el applet `Bienvenido`. El `appletviewer` es un navegador muy básico; sólo sabe cómo interpretar referencias a los applets e ignora todo el demás código HTML en un documento.

Es probable que los programas no funcionen la primera vez. Cada una de las fases anteriores puede fallar, debido a diversos errores que describiremos en este texto. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal para la aritmética con números enteros en Java). Esto haría que el programa de Java imprimiera un mensaje de error. El programador tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir con las fases restantes de nuevo, para determinar que las correcciones funcionen apropiadamente.



Error común de programación 1.1

Los errores, como la división entre cero, ocurren a medida que se ejecuta un programa, de manera que a estos errores se les llama errores en tiempo de ejecución. *Los* errores fatales en tiempo de ejecución *hacen que los programas terminen inmediatamente, sin haber realizado correctamente su trabajo.* Los errores no fatales en tiempo de ejecución *permiten a los programas ejecutarse hasta terminar su trabajo, lo que a menudo produce resultados incorrectos.*

La mayoría de los programas en Java tienen datos de entrada o datos de salida. Cuando decimos que un programa imprime un resultado, generalmente queremos decir que el programa muestra ese resultado en el monitor de la computadora. Los datos pueden enviarse a otros dispositivos, como son discos e impresoras.

1.14 Generalidades acerca de Java y este libro

Java es un poderoso lenguaje de programación. Algunas veces, los programadores experimentados se enorgullecen en poder crear un uso excéntrico, deformado e intrincado de un lenguaje. Ésta es una mala práctica de programación. Hace que los programas sean más difíciles de leer, que se comporten en forma extraña, más difíciles de probar y depurar, y más difíciles de adaptarse a los requerimientos cambiantes.

Este libro está también orientado hacia los programadores principiantes, por lo que nos enfocamos en la *claridad*. La siguiente es nuestra primera "buena práctica de programación".



Buena práctica de programación 1.1

Escriba sus programas de Java en una manera simple y directa. A esto se le conoce como KIS (Keep It Simple, simplifíquelo). No "estire" el lenguaje experimentando con usos excéntricos.

Seguramente habrá escuchado que Java es un lenguaje portable y que los programas escritos en Java pueden ejecutarse en muchas computadoras distintas. Para la programación en general, la *portabilidad* es una *meta elusiva*. El documento del estándar ANSI C[6] contiene una extensa lista de cuestiones relacionadas con la portabilidad, y se han escrito libros completos que hablan sobre la portabilidad.[7, 8]



Tip de portabilidad 1.2

Aunque es más fácil escribir programas portables en Java que en otros lenguajes de programación, las diferencias entre compiladores, intérpretes y computadoras pueden hacer que la portabilidad sea difícil de lograr. No basta con escribir programas en Java para garantizar su portabilidad.

6. ANSI, *American National Standard for Information Systems-Programming Language C (Documento ANSI ANSI/ISO 9899: 1990)*. Nueva York, NY: Instituto nacional estadounidense de estándares, 1990.
7. Jaeschke, R. *Portability and the C Language*. Indianápolis., IN: Hayden Books. 1989.
8. Rabinowitz, H., y C. Schaap. *Portable C*. Englewood Cliffs. NJ: Prentice Hall, 1990.

http://librosysolucionarios.net



Tip para prevenir errores 1.1

Pruebe siempre sus programas de Java en todos los sistemas en los que tenga pensado ejecutarlos, para asegurarse de que trabajen correctamente para las audiencias a las que están destinados.

Comparamos nuestra presentación con la documentación de Java de Sun, para verificar que sea completa y precisa. Sin embargo, como Java es un lenguaje extenso, existen algunos temas que no hemos cubierto. Para obtener detalles técnicos, le sugerimos leer la documentación de Java más actualizada, disponible en el sitio `java.sun.com`. Este libro contiene una extensa bibliografía de libros y artículos sobre Java, y sobre la programación orientada a objetos. Existe una versión basada en Web de la documentación de las APIs de Java en la página `java.sun.com/j2se/1.4.1/docs/api/index.html`. También puede descargar esta documentación en su propio equipo, visitando `java.sun.com/j2se/1.4.1/download.html`.



Buena práctica de programación 1.2

Lea la documentación correspondiente a la versión de Java que esté utilizando. Consulte esta documentación con frecuencia, para conocer la vasta colección de herramientas disponibles en Java, y para asegurarse de que las está utilizando correctamente.



Buena práctica de programación 1.3

Su computadora y su compilador son buenos maestros. Si, después de leer cuidadosamente el manual de documentación de Java, todavía no está seguro de cómo funciona alguna de sus características, experimente y vea lo que ocurre. Analice cada error o mensaje de advertencia que obtenga al compilar sus programas (a estos errores se les llama errores en tiempo de compilación*), y corrija los programas para eliminar estos mensajes.*



Buena práctica de programación 1.4

El Kit de desarrollo de software para Java 2 incluye el código fuente de Java. Muchos programadores leen el código fuente de las clases de las APIs de Java, para determinar cómo funcionan estas clases y para aprender técnicas de programación adicionales.

Este libro explica cómo funciona Java en sus implementaciones actuales. Tal vez el problema más notable con las primeras versiones es que los programas se ejecutaban en forma interpretativa, en el equipo cliente. Los intérpretes se ejecutan lentamente, en comparación con el código máquina totalmente compilado.



Tip de rendimiento 1.2

Los intérpretes tienen una ventaja sobre los compiladores: que un programa interpretado puede empezar su ejecución inmediatamente después de ser descargado en el equipo cliente, mientras que para compilar un programa fuente, éste primero debe pasar por un retraso potencialmente largo durante su compilación, antes de poder ejecutarse.



Tip de portabilidad 1.3

Aunque en los primeros sistemas de Java sólo había intérpretes de Java disponibles para ejecutar códigos de bytes en el sitio del cliente, se han escrito ya compiladores que traducen los códigos de bytes de Java (o en algunos casos, el código fuente de Java) en código máquina nativo del equipo cliente, para la mayoría de las plataformas populares. Estos programas compilados tienen un rendimiento similar al del código de C o C++ compilado. Sin embargo, no hay compiladores de código de bytes para todas las plataformas de Java, por lo que los programas de Java no tendrán el mismo nivel de rendimiento en todas las plataformas.

Los applets presentan algunas cuestiones más interesantes. Recuerde, un applet podría llegar desde casi cualquier servidor Web en el mundo. Por lo tanto, el applet debe tener la capacidad de ejecutarse en cualquier plataforma Java existente.



Tip de rendimiento 1.3

El compilador JIT ("just-in-time", justo a tiempo) es un paso intermedio entre los intérpretes y compiladores, el cual produce código compilado para el programa, a medida que funciona el intérprete y ejecuta los programas en lenguaje máquina, en lugar de reinterpretarlos. Tal vez los compiladores JIT no produzcan lenguaje máquina que sea tan eficiente como el de un compilador optimizador completo.



Tip de rendimiento 1.4

Será conveniente que, para obtener la información más reciente sobre la traducción de programas en Java de alta velocidad, leyera acerca del compilador HotSpot™ de Sun, un componente estándar del Entorno en tiempo de ejecución de Java 2. Visite la página `java.sun.com/products/hotspot`.

El compilador de Java, `javac`, no es un compilador tradicional en cuanto a que no convierte un programa de Java, de código fuente a código máquina nativo para una plataforma computacional específica. En vez de ello, el compilador de Java traduce el código fuente en códigos de bytes; el lenguaje de la Máquina virtual de Java (JVM). La JVM es un programa que simula la operación de una computadora y ejecuta su propio lenguaje máquina (es decir, códigos de bytes de Java).



Observación de ingeniería de software 1.4

Para las organizaciones que quieren desarrollar sistemas de información de uso rudo, existen Entornos integrados de desarrollo (IDEs) disponibles por parte de muchos proveedores de software importantes, incluyendo Sun Microsystems. Los IDEs ofrecen muchas herramientas que dan soporte al proceso de desarrollo de software, como editores para escribir y editar programas, depuradores para localizar errores lógicos en programas y muchas otras características. Algunos de los IDEs más populares son Borland® JBuilder® (`www.borland.com/jbuilder`*), IntelliJ IDEA (*`www.intellij.com/idea`*) y Netbeans (*`www.netbeans.org`*).*



Observación de ingeniería de software 1.5

El poderoso IDE para Java de Sun Microsystems, Inc., Sun™ ONE Studio, Community Edition, viene incluido en el CD que acompaña a este libro, y puede descargarse también de `www.sun.com/software/sundev/jde/index.html`*. Sun también ofrece una versión empresarial de Sun ONE Studio, con un costo, y una versión gratuita para desarrollar aplicaciones móviles con Java 2 Micro Edition (J2ME).*

1.15 Acerca de los objetos: Introducción a la tecnología de objetos y el Lenguaje Unificado de Modelado

Ahora comenzamos con una introducción al tema de la orientación a objetos. La orientación a objetos es una manera natural de pensar acerca del mundo real y de escribir programas de cómputo.

En cada uno de los primeros cinco capítulos nos concentramos en la metodología "convencional" de la programación estructurada, ya que los objetos que creamos utilizan estas piezas de programa estructurado. Sin embargo, terminamos cada uno de esos capítulos con un ejemplo práctico opcional en la sección "Acerca de los objetos", en donde presentamos una introducción cuidadosamente guiada al tema de la orientación a objetos. Nuestro objetivo en estas secciones "Acerca de los objetos" es ayudarle a desarrollar una forma de pensar orientada a objetos, de manera que pueda utilizar inmediatamente las técnicas de programación orientada a objetos que presentamos a partir del capítulo 8. Las secciones "Acerca de los objetos" también lo introducen al *Lenguaje Unificado de Modelado™ (UML™)*. UML es un lenguaje gráfico que permite a las personas que crean sistemas (por ejemplo, arquitectos de software, ingenieros de sistemas, programadores, etcétera.) representar sus requerimientos, análisis y diseños orientados a objetos mediante el uso de una notación común.

En esta sección presentamos los conceptos básicos y la terminología de la orientación a objetos. Las secciones opcionales esparcidas a lo largo de este libro presentan el diseño y la implementación de un simulador de elevador, orientados a objetos. Las secciones opcionales al final de los capítulos 2 al 7 analizan la declaración de un problema típico en el que se requiere la construcción de un sistema, determinan los objetos necesarios para implementar ese sistema, los atributos que necesitan tener estos objetos, los comportamientos que necesitan exhibir estos objetos y especifican la manera en que los objetos necesitan interactuar entre sí para cumplir con los requerimientos del sistema. ¡Y todo esto ocurre *antes* de que usted aprenda a escribir programas de Java orientados a objetos! Las secciones opcionales al final de los capítulos 8, 10 al 14 y 16, modifican y mejoran el diseño presentado en los capítulos 2 al 7. El capítulo 19 presenta la manera de mostrar en pantalla nuestro diseño, rico en multimedia. En los apéndices D a F presentamos una implementación completa en Java del sistema orientado a objetos que diseñamos en los capítulos anteriores.

Este ejemplo práctico le ayudará a prepararse para los proyectos más apegados a la realidad que muy probablemente encontrará en la industria. Si usted es estudiante y su instructor no planea incluir este ejemplo práctico en su curso, considere cubrir este ejemplo práctico por su cuenta. Las secciones del ejemplo práctico refuerzan el material que se cubre en los capítulos correspondientes. Usted experimentará una sólida introducción al diseño orientado a objetos con UML. Además, afinará sus habilidades para leer código al ver paso a paso un programa en Java de 3,320 líneas, cuidadosamente escrito y bien documentado, el cual resulta completamente el programa presentado en el ejemplo práctico.

Comenzaremos nuestra introducción al tema de la orientación a objetos con cierta terminología clave. En cualquier parte del mundo real puede ver *objetos*: gente, animales, plantas, autos, aviones, edificios, computadoras, etc. Los humanos pensamos en términos de objetos. Poseemos la maravillosa habilidad de la *abstracción*, que nos permite ver imágenes digitales de personas, aviones, árboles y montañas como objetos, en vez de verlos como puntos de colores individuales. Si queremos, podemos pensar en términos de playas en vez de granos de arena, en bosques en vez de árboles y en casas en vez de ladrillos.

Podríamos optar por dividir los objetos en dos categorías: objetos animados y objetos inanimados. Los objetos animados están "vivos" en cierto sentido; se mueven a su alrededor y hacen cosas. Por otro lado, los objetos inanimados no parecen hacer mucho, no se mueven por su propia cuenta. Sin embargo, todos esos objetos tienen ciertas cosas en común. Todos ellos tienen *atributos* como tamaño, forma, color y peso; y todos exhiben *comportamientos* (por ejemplo, una pelota rueda, rebota, se infla y desinfla; un bebé llora, duerme, gatea, camina y parpadea; un auto acelera, frena y da vuelta; una toalla absorbe agua).

Los humanos aprenden acerca de los objetos estudiando sus atributos y observando sus comportamientos. Distintos objetos pueden tener atributos similares y pueden exhibir comportamientos similares. Por ejemplo, pueden hacerse comparaciones entre los bebés y los adultos, y entre los humanos y los chimpancés. Los autos, camiones, vagoncitos rojos y patines tienen mucho en común.

El diseño orientado a objetos modela el software en términos similares a los que utilizan las personas para describir objetos del mundo real. Este diseño aprovecha las relaciones entre las *clases*, en donde los objetos de cierta clase (como una clase de vehículos) tienen las mismas características. También aprovecha las relaciones de herencia, e incluso las relaciones de *herencia múltiple*,[9] en donde las nuevas clases de objetos se derivan absorbiendo las características de las clases existentes y agregando sus propias características únicas. Un objeto de la clase "convertible" ciertamente tiene las características de la clase más general "automóvil", además de que el techo de un convertible puede ponerse y quitarse.

El *Diseño orientado a objetos (DOO)* ofrece una manera más natural e intuitiva de ver el proceso de diseño: a saber, *modelando* los componentes de software de igual forma que como describimos los objetos del mundo real (por sus atributos y comportamientos). El DOO también modela la comunicación entre los objetos. Así como las personas se envían mensajes unas a otras (por ejemplo, un sargento ordenando a un soldado que permanezca firme), los objetos también se comunican mediante mensajes.

El DOO *encapsula* los atributos y las *operaciones* (comportamiento) en los *objetos*; los atributos y las operaciones de un objeto se enlazan íntimamente entre sí. Los objetos tienen una propiedad que se conoce como *ocultamiento de información*. Esto significa que, aunque los objetos pueden saber cómo comunicarse entre sí a través de *interfaces* bien definidas, generalmente no se les permite saber cómo se implementan otros objetos; los detalles de la implementación se ocultan dentro de los mismos objetos. Seguramente es posible conducir un auto con efectividad, sin necesidad de saber los detalles acerca de cómo funcionan internamente los motores, las transmisiones y los sistemas de escape. Más adelante veremos por qué el ocultamiento de información es tan imprescindible para la buena ingeniería de software.

Los lenguajes como Java son *orientados a objetos*. La programación en dichos lenguajes se llama *programación orientada a objetos (POO)*, y permite a los diseñadores implementar un diseño orientado a objetos como un sistema funcional. Por otra parte, los lenguajes como C son *por procedimientos*, de manera que la programación tiende a ser *orientada a la acción*. En C, la unidad de programación es la *función*. En Java, la unidad de programación es la clase a partir de la cual se obtienen (crean) la instancias; es decir, los objetos. Las clases en Java contienen *métodos* (que implementan operaciones) y *campos* (que implementan atributos).

Los programadores de C se concentran en escribir funciones. Los grupos de acciones que realizan cierta tarea común se agrupan para formar funciones, y éstas se agrupan para formar programas. Ciertamente los datos son importantes en C, pero éstos existen principalmente para apoyar las acciones que realizan las funciones.

Los programadores de Java se concentran en crear *tipos definidos por el programador*, a los cuales se les llama *clases* e *interfaces*. Cada clase contiene campos y el conjunto de métodos que manipulan esos campos, y que proporcionan servicios a los clientes. El programador utiliza tipos predefinidos como "bloques de cons-

9. Más adelante aprenderemos que aunque Java, a diferencia de C++, no soporta la herencia múltiple, sí ofrece la mayoría de los beneficios clave de esta tecnología mediante el soporte de múltiples "interfaces" por cada clase.

http://librosysolucionarios.net

trucción" para crear nuevas clases. Java se enfoca en las clases (a partir de las cuales creamos objetos) en vez de las funciones.

Las clases son para los objetos lo que los planos de construcción son para las casas. Podemos construir muchas casas a partir de un plano de construcción y, de igual forma, podemos *instanciar* muchos objetos a partir de una clase. Las clases también pueden tener relaciones con otras clases. Por ejemplo, en un diseño orientado a objetos de un banco, la clase "cajero" necesita estar relacionada con la clase "cliente". A estas relaciones se les llama *asociaciones*.

Cuando el software se empaqueta en forma de clases, éstas pueden reutilizarse en sistemas de software posteriores. Los grupos de clases relacionadas se empaquetan comúnmente como *componentes* reutilizables. Así como los corredores de bienes raíces dicen a sus clientes que los tres factores más importantes que afectan el precio de los bienes raíces son "la ubicación, la ubicación y la ubicación", muchas personas en la comunidad de software creen que los tres factores más importantes que afectan el futuro del desarrollo de software son "la reutilización, la reutilización y la reutilización". Reutilizar las clases existentes cuando se crean nuevas clases y programas es un proceso que ahorra tiempo y esfuerzo. La reutilización también ayuda a los programadores a crear sistemas más confiables, ya que las clases y componentes existentes a menudo han pasado por un proceso extenso de prueba y depuración.

Evidentemente, con la tecnología de objetos podemos crear la mayor parte del software que necesitaremos mediante la combinación de clases, que son "partes intercambiables". Cada nueva clase que usted cree tendrá el potencial de convertirse en una valiosa pieza de software, que usted y otros programadores podrán usar para agilizar y mejorar la calidad de los futuros esfuerzos de desarrollo de software.

Introducción al análisis y diseño orientados a objetos (A/DOO)

Pronto estará escribiendo programas en Java. ¿Cómo creará el código para sus programas? Si usted es como muchos programadores principiantes, simplemente encenderá su computadora y empezará a teclear. Esta metodología puede funcionar para pequeños proyectos, pero ¿qué haría usted si se le pidiera crear un sistema de software para controlar las máquinas de cajero automático para un importante banco? Dicho proyecto sería demasiado largo y complejo como para que usted se sentara y simplemente empezara a teclear.

Para crear las mejores soluciones, debe seguir un proceso detallado para obtener un *análisis* de los requerimientos de su proyecto, y para desarrollar un *diseño* que cumpla con esos requerimientos. Idealmente usted pasaría por este proceso y revisaría el diseño antes de escribir el código para su proyecto. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, lo llamamos un proceso de *análisis y diseño orientado a objetos (A/DOO)*. Los programadores experimentados saben que, sin importar qué tan simple parezca un problema, el análisis y el diseño pueden ahorrarnos innumerables horas que podrían perderse al abandonar, en plena implementación, un método de desarrollo de un sistema mal planeado.

A/DOO es el término genérico para el proceso de analizar un problema y desarrollar un método para resolverlo. Los pequeños problemas como los que se describen en los primeros capítulos de este libro no requieren de un proceso exhaustivo. Podría ser suficiente con escribir *pseudocódigo* antes de empezar a escribir el código. El pseudocódigo es un medio informal de expresar el código de un programa. En realidad no es un lenguaje de programación, pero podemos usarlo como un tipo de "bosquejo" para guiarnos a medida que escribimos nuestro código. En el capítulo 4 presentamos el pseudocódigo.

El pseudocódigo puede ser suficiente para los problemas pequeños, pero a medida que éstos y los grupos de personas que los resuelven se incrementan en tamaño, los métodos de A/DOO se vuelven más necesarios. Idealmente, un grupo debería acordar un proceso estrictamente definido para resolver su problema, y acordar también una manera uniforme para que los miembros del grupo se comuniquen los resultados de ese proceso entre sí. Aunque existen muchos procesos de A/DOO distintos, existe un lenguaje gráfico para comunicar los resultados de cualquier proceso A/DOO que se ha vuelto muy popular. Este lenguaje se conoce como *Lenguaje Unificado de Modelado (UML)*. UML se desarrolló a mediados de la década de los noventa, bajo la dirección inicial de tres metodologistas de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

Historia de UML

En la década de los ochenta, un creciente número de empresas comenzó a utilizar la POO para programar sus aplicaciones, lo cual generó la necesidad de un proceso establecido con el cual acercarse al A/DOO. Muchos metodologistas (incluyendo a Booch, Rumbaugh y Jacobson) produjeron y promocionaron, por su cuenta, procesos separados para satisfacer esta necesidad. Cada uno de estos procesos tenía su propia notación, o "lenguaje" (en forma de diagramas gráficos), para transmitir los resultados del análisis y el diseño.

A principios de la década de los noventa, diversas compañías (e inclusive diferentes divisiones dentro de la misma compañía) utilizaban distintos procesos y notaciones. Además, estas compañías querían utilizar herramientas de software que tuvieran soporte para procesos particulares. Con tantos procesos, se les dificultó a los distribuidores de software proporcionar dichas herramientas. Evidentemente era necesario contar con una notación y procesos estándar.

En 1994, James Rumbaugh se unió con Grady Booch en Rational Software Corporation, y los dos comenzaron a trabajar para unificar sus populares procesos. Pronto se unió a ellos Ivar Jacobson. En 1996, el grupo liberó las primeras versiones de UML para la comunidad de ingeniería de software, solicitando retroalimentación. Casi al mismo tiempo, una organización conocida como *Object Management Group™* (*OMG™*, Grupo de administración de objetos) hizo una invitación para participar en la creación de un lenguaje común de modelado. El OMG es una organización sin fines de lucro que promueve el uso de la tecnología orientada a objetos, emitiendo lineamientos y especificaciones para las tecnologías orientadas a objetos. Varias empresas (entre ellas HP, IBM, Microsoft, Oracle y Rational Software) habían reconocido ya la necesidad de un lenguaje común de modelado. Estas compañías formaron el consorcio *UML Partners (Socios de UML)* en respuesta a la solicitud de proposiciones por parte del OMG. Este consorcio desarrolló la versión 1.1 de UML, y la envió al OMG. El OMG aceptó la proposición y, en 1997, asumió la responsabilidad del mantenimiento y revisión de UML en forma continua. En el 2001, el OMG liberó UML versión 1.4 (la versión actual al momento de escribir este libro) y está trabajando en la versión 2.0 (programado tentativamente para su liberación en el 2003).

¿Qué es UML?

El Lenguaje Unificado de Modelado es ahora el esquema de representación gráfica más utilizado para modelar sistemas orientados a objetos. Evidentemente ha unificado los diversos esquemas de notación populares. Aquellos quienes diseñan sistemas utilizan el lenguaje (en forma de diagramas) para modelar sus sistemas.

Una de las características más atractiva de UML es su flexibilidad. UML es extensible e independiente de los diversos procesos de A/DOO. Los modeladores de UML tienen la libertad de diseñar sistemas utilizando varios procesos, pero todos los desarrolladores pueden ahora expresar esos diseños con un conjunto de notaciones estándar.

UML es un lenguaje gráfico complejo, con muchas características. En nuestras secciones "Acerca de los objetos", presentamos un subconjunto conciso y simplificado de estas características. Luego utilizamos este subconjunto para guiar al lector a través de la experiencia de su primer diseño con UML, la cual está dirigida al diseñador/programador principiante orientado a objetos. Para obtener una explicación más completa acerca de UML, consulte el sitio Web Object Management Group (Grupo de administración de objetos) (`www.omg.org`) y el documento de especificación oficial para UML 1.4 (`www.omg.org/uml`). Además se han publicado muchos libros acerca de UML. Rational Software Corporation proporciona una lista de lecturas recomendadas sobre libros de UML en la página `rational.com/uml/reading/index.jsp`. Los libros se dividen en los niveles básico, intermedio y avanzado. También puede visitar la página de Pearson en `www.pearsoneducacion.net` o algún sitio de venta en línea de libros, como `www.amazon.com` y busque con el término UML.

1.16 Descubrimiento de patrones de diseño: Introducción

En esta sección empieza nuestro análisis de los patrones de diseño, y se titula Descubrimiento de patrones de diseño". La mayoría de los ejemplos que se proporcionan en este libro son relativamente pequeños. Estos ejemplos no requieren de un extenso proceso de diseño, ya que utilizan sólo unas cuantas clases e ilustran los conceptos de programación introductorios. Sin embargo, algunos programas, como el ejemplo práctico de nuestra simulación de elevador, son más complejos: pueden requerir miles de líneas de código o incluso más, pueden contener muchas interacciones entre los objetos e implicar muchas interacciones por parte de los usuarios. Los sistemas más grandes, como las máquinas de cajero automático o los sistemas de control del tráfico aéreo, pueden contener millones de líneas de código. El diseño efectivo es imprescindible para una construcción apropiada de dichos sistemas complejos.

http://librosysolucionarios.net

Durante la década pasada, la industria de la ingeniería de software progresó considerablemente en el campo de los *patrones de diseño*: arquitecturas comprobadas para construir software orientado a objetos que sea flexible y pueda mantenerse.[10] Al utilizar patrones de diseño se puede reducir considerablemente la complejidad del proceso de diseño. Diseñar un sistema de ATM será una tarea mucho menos pesada si los desarrolladores utilizan patrones de diseño. Los patrones de diseño benefician a los desarrolladores de sistemas al

- ayudar a crear software confiable, utilizando arquitecturas comprobadas y la experiencia acumulada en la industria,
- promover la reutilización del diseño en sistemas posteriores,
- ayudar a identificar los errores y obstáculos comunes que ocurren al crear sistemas,
- ayudar a diseñar sistemas, independientemente del lenguaje en el que se vayan a implementar,
- establecer un vocabulario de diseño común entre los desarrolladores, y
- acortar la fase de diseño en un proceso de desarrollo de software.

La noción de utilizar patrones de diseño para crear sistemas de software se originó en el campo de la arquitectura. Al diseñar edificios, los arquitectos utilizan un conjunto de elementos de diseño arquitectónico establecidos, como arcos y columnas. Diseñar con arcos y columnas es una estrategia comprobada para construir edificios sólidos; estos elementos pueden verse como patrones de diseño arquitectónico.

En el software, los patrones de diseño no son clases ni objetos. Los diseñadores utilizan patrones de diseño para construir conjuntos de clases y objetos. Para utilizar los patrones de diseño con efectividad, los diseñadores deben familiarizarse a sí mismos con los patrones más populares y efectivos que se utilicen en la industria de la ingeniería de software. En este libro hablamos sobre las arquitecturas y patrones de diseño fundamentales orientados a objetos, así como de su importancia para construir software con buena ingeniería.

A lo largo del libro presentamos varios patrones de diseño en Java, pero éstos pueden implementarse en cualquier lenguaje orientado a objetos como C++, o Visual Basic. Describimos varios patrones de diseño utilizados por Sun Microsystems en la API de Java. Utilizamos patrones de diseño en muchos programas de este libro, los cuales identificaremos a lo largo de nuestra discusión. Estos programas proporcionan ejemplos acerca del uso de patrones de diseño para construir software orientado a objetos que sea confiable y robusto.

Historia de los patrones de diseño orientados a objetos

De 1991 a 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (conocidos colectivamente como la "Banda de los cuatro") utilizaron su experiencia combinada para escribir el libro *Design Patterns: Elements of Reusable Object-Oriented Software*. Este libro describe 23 patrones de diseño, cada uno de los cuales proporciona una solución para un problema común de diseño de software en la industria. El libro agrupa los patrones de diseño en tres categorías: *patrones de diseño de creación*, *patrones de diseño de estructura* y *patrones de diseño de comportamiento*. Los patrones de diseño de creación describen técnicas para instanciar objetos (o grupos de objetos). Los patrones de diseño de estructura permiten a los diseñadores organizar clases y objetos en estructuras más grandes. Los patrones de diseño de comportamiento asignan responsabilidades a los objetos.

El libro de la banda de los cuatro demostró que los patrones de diseño evolucionaron naturalmente, a través de años de experiencia en la industria. En su artículo *Seven Habits of Successful Pattern Writers*[11], John Vlissides declara que "la actividad más importante en la escritura de patrones es la reflexión". Esta declaración implica que, para crear patrones, los desarrolladores deben reflexionar sobre sus éxitos (y errores), y documentarlos. Los desarrolladores utilizan patrones de diseño para capturar y emplear esta experiencia colectiva en la industria, lo que en última instancia les ayuda a evitar cometer los mismos errores dos veces. Se están creando nuevos patrones de diseño todo el tiempo, y se introducen rápidamente a los diseñadores en todo el mundo mediante Internet.

Nuestro análisis de los patrones de diseño empieza aquí, y termina con cinco secciones opcionales de "Descubrimiento de los patrones de diseño" al final de los capítulos 10, 14, 16, 18 y 22. Cada una de estas secciones se coloca al final del capítulo que introduce las tecnologías necesarias de Java. Si usted es estudiante y su instructor no planea incluir este material en su curso, le sugerimos que lo lea por cuenta propia.

10. Gamma, E., R. Helm, R. Johnson y J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
11. Vlissides, J. *Pattern Hatching: Design Patterns Applied*. Reading, MA: Addison-Wesley, 1998.

http://librosysolucionarios.net

RESUMEN

- Una computadora está compuesta por diversos dispositivos (como el teclado, monitor, discos, memoria y unidades de procesamiento) conocidos como hardware.
- A los programas de cómputo que se ejecutan en una computadora se les denomina *software*.
- Java es uno de los lenguajes para desarrollo de software más populares en la actualidad. Java es un lenguaje completamente orientado a objetos, el cual ofrece un sólido soporte para las técnicas apropiadas de ingeniería de software.
- Java fue desarrollado por Sun Microsystems. Sun ofrece una implementación de la Plataforma Java 2, Standard Edition conocida como el Kit de desarrollo de software para Java 2 (J2SDK), versión 1.4.1, que incluye el conjunto mínimo de herramientas que se necesitan para escribir software en Java.
- Una computadora es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de millones (incluso de miles de millones) de veces más rápidas que los humanos.
- Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas programas de cómputo. Los programas de cómputo guían a la computadora a través de conjuntos de acciones especificadas por los programadores de computadoras.
- La unidad de entrada es la sección "receptora" de la computadora. Obtiene información de los dispositivos de entrada y la coloca a disposición de las otras unidades, para su procesamiento.
- La unidad de salida es la sección de "embarque" de la computadora. Ésta toma información que ya ha sido procesada por la computadora y la coloca en los dispositivos de salida, para que esté disponible fuera de la computadora.
- La unidad de memoria es la sección que funciona en la computadora como un "almacén" de acceso rápido, pero con relativa baja capacidad. Retiene la información que se ha introducido a través de la unidad de entrada, de forma que la información pueda estar inmediatamente disponible para su procesamiento, cuando sea necesario, y retiene la información que ya ha sido procesada hasta que la unidad de salida pueda colocarla en los dispositivos de salida.
- La unidad aritmética y lógica (ALU) es la sección de "manufactura" de la computadora. Es responsable de realizar cálculos y de tomar decisiones.
- La unidad central de procesamiento (CPU) es la sección "administrativa" de la computadora. Es quien coordina y supervisa la operación de las demás secciones.
- La unidad de almacenamiento secundario es el "almacén" de alta capacidad y de larga duración de la computadora. Los programas o datos que no están utilizando las otras unidades, normalmente se colocan en dispositivos de almacenamiento secundario (como discos) hasta que son requeridos de nuevo, posiblemente horas, días, meses o incluso años después.
- Los sistemas de software denominados sistemas operativos se desarrollaron para facilitar el uso de la computadora. Los primeros sistemas operativos se encargaban de la transición uniforme entre trabajos y minimizaban el tiempo requerido para que los operadores de computadoras se alternaran entre los trabajos.
- La multiprogramación significa la operación "simultánea" de muchas tareas dentro de la computadora (ésta comparte sus recursos entre los trabajos que compiten por su atención).
- El tiempo compartido es un caso especial de la multiprogramación, en el cual los usuarios acceden a la computadora a través de terminales. La computadora ejecuta una pequeña porción del trabajo de un usuario y después procede a dar servicio al siguiente usuario. La computadora hace el trabajo tan rápidamente que puede proporcionar el servicio a cada usuario varias veces por segundo, de manera que los programas parecen ejecutarse en forma simultánea.
- En 1977, Apple Computer popularizó el fenómeno de la computación personal. En 1981, IBM introdujo la PC (Computadora personal) de IBM, lo que legitimó inmediatamente la computación personal en las organizaciones de negocios, de la industria y del gobierno.
- Con la computación distribuida, la computación de una empresa se distribuye mediante redes a los sitios en donde se realiza el trabajo de la empresa. Actualmente la información se comparte fácilmente a través de redes de computadoras, en donde los servidores de archivos ofrecen un almacén común para programas y datos, el cual puede ser utilizado por computadoras cliente distribuidas en toda la red; de ahí el término de computación cliente/servidor.
- Java se ha convertido en el lenguaje de elección para desarrollar aplicaciones basadas en Internet (y para muchos otros propósitos).
- Los lenguajes de computación pueden dividirse en tres tipos generales: lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel.
- Cualquier computadora puede entender de manera directa sólo su propio lenguaje máquina. Por lo general, los lenguajes máquina consisten en cadenas de números [que finalmente se reducen a unos (1) y ceros (0)] que instruyen a las computadoras para realizar sus operaciones más elementales, una por una. Los lenguajes máquina son dependientes del equipo.

- Las abreviaturas del inglés forman la base de los lenguajes ensambladores. Los programas traductores conocidos como ensambladores convierten los programas en lenguaje ensamblador a lenguaje máquina.
- Los compiladores traducen los programas en lenguaje de alto nivel a programas en lenguaje máquina. Los lenguajes de alto nivel (como Java) contienen palabras en inglés y notaciones matemáticas convencionales.
- Los programas intérpretes ejecutan programas en lenguaje de alto nivel directamente, sin necesidad de compilarlos en lenguaje máquina.
- Los objetos son, esencialmente, componentes reutilizables de software que se modelan en términos como los que utilizamos para describir las cosas en el mundo real.
- Java se originó en Sun Microsystems como un proyecto para los dispositivos electrodomésticos inteligentes. Java se utiliza ahora para crear páginas Web con contenido dinámico e interactivo, para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores Web, para proporcionar aplicaciones a los dispositivos domésticos, etc.
- Los programas en Java consisten de varias piezas que se llaman *clases*. Estas clases consisten de piezas llamadas métodos, los cuales realizan tareas y devuelven información cuando completan esas tareas. La mayoría de los programadores de Java utilizan las vastas colecciones de clases existentes en las bibliotecas de clases de Java.
- A principios de la década de los ochenta, Bjarne Stroustrup desarrolló una extensión de C en los laboratorios Bell: C++. Este lenguaje proporciona un conjunto de características que "pulen" al lenguaje C pero, lo más importante es que proporciona la capacidad de una programación orientada a objetos.
- FORTRAN (FORmula TRANslator) fue desarrollado por IBM Corporation en la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería, que requerían cálculos matemáticos complejos.
- COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado en 1959 por un grupo de fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria. COBOL se utiliza principalmente en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos.
- Pascal se diseñó aproximadamente al mismo tiempo que C. Fue creado por el profesor Niklaus Wirth para uso académico.
- Basic fue desarrollado en 1965 en el Dartmouth College, como un lenguaje simple para ayudar a los principiantes a familiarizarse con la programación.
- Ada se desarrolló bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DOD) durante la década de los setenta y los primeros años de la década de los ochenta. Una de las características importantes de Ada se conoce como multitarea; ésta permite a los programadores especificar que muchas actividades ocurrirán en paralelo.
- El lenguaje de programación BASIC (Beginner's All-Purpose Symbolic Instruction Code, Código de instrucciones simbólicas de uso general para principiantes) fue desarrollado a mediados de la década de los sesenta por los profesores John Kemeny y Thomas Kurtz, del Dartmouth College, como un lenguaje para escribir programas simples. El propósito principal de BASIC era que los principiantes se familiarizaran con las técnicas de programación. Visual Basic se introdujo en 1991 para simplificar el desarrollo de aplicaciones para Microsoft Windows.
- De manera similar a Java, la plataforma .NET permite la distribución de aplicaciones basadas en Web a muchos dispositivos (incluso teléfonos celulares) y computadoras de escritorio.
- Visual Basic .NET está diseñado para la nueva plataforma de programación .NET de Microsoft. Visual Basic .NET ofrece una orientación a objetos mejorada en comparación con las primeras versiones de Visual Basic, además de que hace uso de la poderosa biblioteca de .NET, conocida como Biblioteca de clases del Framework (FCL).
- Visual C++ es una implementación de Microsoft de C++, que incluye extensiones propias de Microsoft para el lenguaje. En un principio, la programación de gráficos y GUI en Visual C++ se implementaba mediante la MFC (Microsoft Foundation Classes). Ahora, con la introducción de .NET, la FCL se comparte entre Visual Basic, Visual C++, C# (el nuevo lenguaje de Microsoft) y muchos otros lenguajes que Microsoft y otros distribuidores están poniendo a disposición de .NET.
- El lenguaje de programación C#, desarrollado en Microsoft por un equipo dirigido por Anders Hejlsberg y Scott Wiltamuth, fue diseñado específicamente para la plataforma .NET como un lenguaje que permitiera a los programadores migrar fácilmente a .NET.
- Java, a través de una técnica que se conoce como subprocesamiento múltiple, permite a los programadores escribir programas con actividades paralelas.
- Internet se desarrolló hace más de cuatro décadas, y su patrocinio estuvo a cargo del Departamento de Defensa de los Estados Unidos. Actualmente, Internet es utilizada por cientos de millones de computadoras en todo el mundo.

- Web permite a los usuarios de computadora ver documentos con uso intensivo de multimedia a través de Internet.
- Los nombres de archivos de código fuente en Java terminan con la extensión `.java`.
- El compilador de Java (`javac`) traduce un programa en Java a códigos de bytes: el lenguaje que el intérprete de Java puede entender. Si un programa se compila correctamente, el compilador produce un archivo con la extensión `.class`. Éste es el archivo que contiene los códigos de bytes que se interpretarán durante la fase de ejecución.
- Un programa en Java debe colocarse en memoria antes de que pueda ejecutarse. De esto se encarga el cargador de clases, que toma el archivo (o archivos) `.class` que contiene los códigos de bytes y lo transfiere a la memoria principal. El archivo `.class` puede cargarse desde un disco en su sistema o a través de una red.
- Una aplicación es un programa que generalmente se guarda y se ejecuta en el equipo local del usuario. Las aplicaciones se cargan en memoria y después se ejecutan mediante el intérprete `java`.
- Un applet es un programa que generalmente se guarda en un equipo remoto, al cual se conectan los usuarios mediante un navegador Web. Los applets se cargan desde un equipo remoto en el navegador para ejecutarse.
- Los navegadores se utilizan para ver documentos HTML en World Wide Web. Cuando un navegador encuentra un applet en un documento HTML, el navegador inicia el cargador de clases de Java para cargar el applet. Los navegadores con soporte para Java cuentan con intérpretes de Java integrados. Una vez que se carga el applet, el intérprete de Java en el navegador empieza a ejecutarlo.
- Los applets pueden ejecutarse también desde la línea de comandos, utilizando el comando `appletviewer` que se incluye en el Kit de desarrollo de software para Java 2 (J2SDK). El `appletviewer` se conoce comúnmente como el navegador mínimo; sólo sabe cómo interpretar applets.
- Antes de que los códigos de bytes en un applet se ejecuten mediante el intérprete de Java integrado en un navegador, o mediante el `appletviewer`, éstos se verifican mediante el verificador de código de bytes para asegurar que los códigos de bytes de las clases descargadas sean válidos, y que no violen las restricciones de seguridad de Java.
- El compilador JIT ("just-in-time", justo a tiempo) es un paso intermedio entre los intérpretes y compiladores, el cual produce código compilado para el programa, a medida que funciona el intérprete y ejecuta los programas en lenguaje máquina, en lugar de reinterpretarlos. Los compiladores JIT no producen lenguaje máquina que sea tan eficiente como el de un compilador completo.
- Para las organizaciones que quieren desarrollar sistemas de información de uso rudo, existen Entornos integrados de desarrollo (IDEs) disponibles por parte de muchos proveedores de software importantes. Los IDEs proporcionan muchas herramientas para dar soporte al proceso de desarrollo de software.
- La orientación a objetos es una manera natural de pensar acerca del mundo y de escribir programas de cómputo.
- El Lenguaje Unificado de Modelado (UML) es un lenguaje gráfico, el cual permite a las personas que desarrollan sistemas representar sus diseños orientados a objetos en una notación común.
- Los humanos pensamos en términos de objetos. La abstracción nos permite ver imágenes en pantalla como personas, aviones, árboles y montañas, en vez de verlos como puntos de colores individuales (llamados píxeles, o "elementos de imágenes"). Los humanos aprenden acerca de los objetos estudiando sus atributos y observando sus comportamientos. Distintos objetos pueden tener atributos similares y pueden exhibir comportamientos similares.
- El diseño orientado a objetos (DOO) modela los componentes de software en términos de objetos del mundo real. Este diseño aprovecha las relaciones entre las clases, en donde los objetos de cierta clase tienen las mismas características. También aprovecha las relaciones de herencia, e incluso las relaciones de herencia múltiple, en donde las clases de objetos recién creadas se derivan absorbiendo las características de las clases existentes y agregando sus propias características únicas. El DOO encapsula los datos (atributos) y las funciones (comportamiento) en los objetos; los datos y las funciones de un objeto están estrechamente entrelazados.
- Los objetos tienen una propiedad que se conoce como ocultamiento de información; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, generalmente no se les permite saber cómo están implementados los otros objetos.
- La programación orientada a objetos (POO) permite a los programadores implementar los diseños orientados a objetos en forma de sistemas funcionales.
- En Java, la unidad de programación es la clase a partir de la cual se obtienen las instancias; es decir, los objetos. Los programadores de Java se concentran en crear sus clases. Cada clase contiene datos y funciones para manipular a esos datos. Los componentes de los datos se llaman campos. Los componentes de las funciones se llaman métodos.
- Una instancia de una clase se le conoce como objeto.
- Las clases también pueden tener relaciones con otras clases. A estas relaciones se les llama asociaciones.

http://librosysolucionarios.net

- Con la tecnología de objetos podemos crear la mayor parte del software que necesitaremos mediante la combinación de "partes intercambiables", a las que se les conoce como clases.
- Al proceso de analizar y diseñar su sistema desde un punto de vista orientado a objetos se le conoce como análisis y diseño orientado a objetos (A/DOO).
- El Lenguaje Unificado de Modelado (UML) es ahora el esquema de representación gráfica más ampliamente utilizado para modelar sistemas orientados a objetos. Aquellos quienes diseñan sistemas utilizan el lenguaje (en forma de diagramas gráficos) para modelar sus sistemas.
- Durante la década pasada, la industria de la ingeniería de software progresó considerablemente en el campo de los patrones de diseño: arquitecturas comprobadas para construir software orientado a objetos que sea flexible y pueda mantenerse. El uso de patrones de diseño puede reducir considerablemente la complejidad del proceso de diseño.
- Los patrones de diseño benefician a los desarrolladores de sistemas, ya que les ayudan a construir software confiable, utilizando arquitecturas comprobadas y la experiencia acumulada en la industria, promoviendo la reutilización del diseño en sistemas posteriores, identificando los errores y obstáculos comunes que se presentan al crear sistemas, ayudando a diseñar sistemas independientemente del lenguaje en el que se vayan a implementar, estableciendo un vocabulario de diseño común entre los desarrolladores y acortando la fase de diseño en el proceso de desarrollo de software.
- Los diseñadores utilizan patrones de diseño para construir conjuntos de clases y objetos.
- Los patrones de diseño de creación describen técnicas para instanciar objetos (o grupos de objetos).
- Los patrones de diseño de estructura permiten a los diseñadores organizar clases y objetos en estructuras más grandes.
- Los patrones de diseño de comportamiento asignan responsabilidades a los objetos.

TERMINOLOGÍA

abstracción
Ada
ALU (Unidad aritmética y lógica)
análisis y diseño orientado a objetos (A/DOO)
ANSI C
aplicación
applet
atributo
Basic
bibliotecas de clases
C
C#
C++
cargador de clases
clase
`.class`, archivo
COBOL
código de bytes
comando `appletviewer`
compilador
compilador HotSpot™
compilador JIT (justo a tiempo)
componentes reutilizables
comportamiento
computación cliente/servidor
computación distribuida
computación personal
computadora
contenido dinámico
CPU (unidad central de procesamiento)
dependiente del equipo
disco
diseño orientado a objetos (DOO)
dispositivo de entrada
dispositivo de salida
documento de requerimientos
editor
encapsulamiento
entrada/salida (E/S)
enunciado del problema
error en tiempo de compilación
error en tiempo de ejecución
error fatal en tiempo de ejecución
error no fatal en tiempo de ejecución
fase de carga
fase de compilación
fase de edición
fase de ejecución
fase de verificación
FORTRAN
hardware
herencia
HTML (Lenguaje de Marcado de Hipertexto)
IDE (Entorno Integrado de Desarrollo)
independiente del equipo
Internet
intérprete
Java
`.java`, extensión de nombre de archivo
`java`, intérprete
`javac`, compilador
KIS (simplifíquelo)
Kit de desarrollo de software para Java 2 (J2SDK)
lenguaje de alto nivel
lenguaje ensamblador
lenguaje máquina
Lenguaje Unificado de Modelado™ (UML)
LIVE-CODE™, metodología
Máquina virtual de Java (JVM)
memoria principal
método
Microsoft Internet Explorer, navegador Web
modelado
multiprocesador
multitarea
.NET
Netscape, navegador Web
objeto
Ocultamiento de información
Pascal
patrón de diseño
patrón de diseño de comportamiento
patrón de diseño de creación
patrón de diseño de estructuras
plataforma
portabilidad
programa de cómputo
programa traductor
programación orientada a objetos (POO)
programación por procedimientos
programador de computadoras
reutilización de software
servidor de archivos
sistemas heredados
software
subprocesamiento múltiple
Sun Microsystems
tiempo compartido
unidad central de procesamiento (CPU)
unidad de almacenamiento secundario
unidad de entrada
unidad de memoria
unidad de salida
verificador de código de bytes
Visual Basic .NET
Visual C++.NET
World Wide Web

EJERCICIOS DE AUTOEVALUACIÓN

1.1 Complete las siguientes oraciones:

a) La compañía que popularizó la computación personal fue _____.
b) La computadora que legitimó la computación personal en los negocios y la industria fue _____.
c) Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas _____.
d) Las seis unidades lógicas clave de la computadora son _____, _____, _____, _____, _____ y _____.
e) Los tres tipos de lenguajes descritos en este capítulo son _____, _____ y _____.
f) Los programas que traducen programas en lenguaje de alto nivel a lenguaje máquina se denominan _____.
g) La _____ permite a los usuarios de computadora localizar y ver documentos basados en multimedia sobre casi cualquier tema, a través de Internet.
h) Las _____ de Java generalmente se guardan en su computadora, y están diseñadas para ejecutarse independientemente de los navegadores Web.
i) El _____ permite que un applet o aplicación realice múltiples actividades en paralelo.

1.2 Complete cada una de las siguientes oraciones relacionadas con el entorno de Java:

a) El comando _____ del Kit de desarrollo de software para Java 2 ejecuta un applet.
b) El comando _____ del Kit de desarrollo de software para Java 2 ejecuta una aplicación.
c) El comando _____ del Kit de desarrollo de software para Java 2 compila un programa de Java.
d) Se requiere un documento _____ para invocar a un applet de Java.
e) Un archivo de programa de Java debe terminar con la extensión _____.
f) Cuando se compila un programa en Java, el archivo producido por el compilador termina con la extensión _____.
g) El archivo producido por el compilador de Java contiene _____ que se interpretan para ejecutar un applet o aplicación de Java.

1.3 Complete cada una de las siguientes oraciones (basándose en las secciones 1.15 y 1.16):

a) Durante la década pasada, la industria de la ingeniería de software ha progresado considerablemente en el campo de los _____: arquitecturas comprobadas para construir software orientado a objetos que sea flexible y pueda mantenerse.
b) Los objetos tienen una propiedad que se conoce como _____; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, generalmente no se les permite saber cómo están implementados los otros objetos.
c) Los programadores de Java se concentran en crear sus propios tipos definidos por el usuario, a los cuales se les llama _____.
d) Las clases también pueden tener relaciones con otras clases. A estas relaciones se les llama _____.
e) Al proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se le conoce como _____.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

1.1 a) Apple. b) PC de IBM. c) Programas. d) Unidad de entrada, unidad de salida, unidad de memoria, unidad aritmética y lógica, unidad central de procesamiento, unidad de almacenamiento secundario. e) Lenguajes máquina, lenguajes ensambladores, lenguajes de alto nivel. f) Compiladores. g) World Wide Web. h) Aplicaciones. i) Subprocesamiento múltiple.

1.2 a) `Appletviewer`. b) `Java`. c) `Javac`. d) HTML. e) `.Java`. f) `.Class`. g) Códigos de bytes.

1.3 a) Patrones de diseño. b) Ocultamiento de información. c) Clases. d) Asociaciones. e) Análisis y diseño orientado a objetos (A/DOO).

http://librosysolucionarios.net

EJERCICIOS

1.4 Clasifique cada uno de los siguientes elementos, ya sea como hardware o software.
a) CPU
b) compilador de Java
c) intérprete de Java
d) unidad de entrada
e) editor

1.5 Complete cada una de las siguientes oraciones:
a) La unidad lógica de la computadora que recibe información desde el exterior de la computadora para que ésta la utilice se llama _____.
b) El proceso de indicar a la computadora cómo resolver problemas específicos se llama _____.
c) _____ es un tipo de lenguaje computacional que utiliza abreviaturas del inglés para las instrucciones de lenguaje máquina.
d) _____ es una unidad lógica de la computadora que envía información, que ya ha sido procesada por la computadora, a varios dispositivos, de manera que la información pueda utilizarse fuera de la computadora.
e) _____ es una unidad lógica de la computadora que retiene información.
f) _____ es una unidad lógica de la computadora que realiza cálculos.
g) _____ es una unidad lógica de la computadora que toma decisiones lógicas.
h) Los lenguajes _____ son los más convenientes para que el programador pueda escribir programas rápida y fácilmente.
i) Al único lenguaje que una computadora puede entender directamente se le conoce como el _____ de esa computadora.
j) _____ es una unidad lógica de la computadora que coordina las actividades de todas las demás unidades lógicas.

1.6 Indique la diferencia entre los términos error fatal y error no fatal. ¿Por qué sería preferible experimentar un error fatal, en vez de un error no fatal?

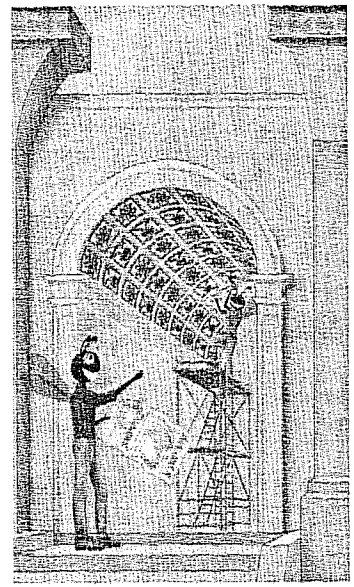
1.7 Use su navegador Web para visitar los siguientes sitios Web, y familiarícese con los recursos de Java disponibles para usted en World Wide Web:
a) `java.sun.com`
b) `java.sun.com/applets`
c) `developer.java.sun.com/developer` [*Nota:* Tal vez necesite registrarse para acceder a este sitio. Sin embargo, el registro es gratuito.]
d) `www.javalobby.org`
e) `www.jguru.com`
f) `www.javaworld.com`
g) `www.fawcette.com/javapro`

1.8 Complete cada una de las siguientes oraciones (basándose en las secciones 1.15 y 1.16):
a) Los patrones de diseño _____ describen técnicas para instanciar objetos (o grupos de objetos).
b) El _____ es ahora el esquema de representación gráfica más ampliamente utilizado para modelar sistemas orientados a objetos.
c) Las clases en Java contienen _____ (que implementan comportamientos) y _____ (que implementan los datos de una clase).
d) Los patrones de diseño _____ permiten a los diseñadores organizar clases y objetos en estructuras más grandes.
e) Los patrones de diseño _____ asignan responsabilidades a los objetos.
f) Los objetos se instancian a partir de _____.

1.9 ¿Por qué es importante estudiar patrones de diseño?

2

Introducción a las aplicaciones en Java



Objetivos

- Escribir aplicaciones simples en Java.
- Utilizar las instrucciones de entrada y salida.
- Familiarizarse con los tipos primitivos.
- Comprender los conceptos básicos de la memoria.
- Utilizar los operadores aritméticos.
- Comprender la precedencia de los operadores aritméticos.
- Escribir instrucciones para tomar decisiones.
- Utilizar los operadores relacionales y de igualdad.

Los comentarios van y vienen, pero los hechos son irrevocables.
C. P. Scott

El acreedor tiene mejor memoria que el deudor.
James Howell

Al hacer frente a una decisión, siempre me pregunto, "¿Cuál será la solución más divertida?"
Peggy Walker

Él ha dejado su cuerpo a la ciencia; y la ciencia está rebatiendo su voluntad.
David Frost

La igualdad, en un sentido social, puede dividirse en condiciones y derechos.
James Fenimore Cooper

http://librosysolucionarios.net

Plan general

2.1 Introducción
2.2 Su primer programa en Java: Imprimir una línea de texto
2.3 Modificación de nuestro primer programa en Java
2.4 Cómo mostrar texto en un cuadro de diálogo
2.5 Otra aplicación en Java: Suma de enteros
2.6 Conceptos acerca de la memoria
2.7 Aritmética
2.8 Toma de decisiones: Operadores de igualdad y relacionales
2.9 (Ejemplo práctico opcional) Acerca de los objetos: Cómo examinar el enunciado de un problema

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

2.1 Introducción

Ahora presentaremos la programación en Java, para lo cual emplearemos ejemplos que ilustren varias características importantes de este lenguaje. Analizaremos cada uno de los ejemplos, línea por línea. Tanto éste como el capítulo 3 presentan dos tipos de programas en Java: *aplicaciones* y *applets*. En los capítulos 4 y 5 presentaremos un análisis detallado acerca del *desarrollo de programas* y el *control de programas* en Java.

2.2 Su primer programa en Java: Imprimir una línea de texto

Cada vez que utiliza una computadora, ejecuta diversas aplicaciones que realizan tareas por usted. Por ejemplo, su aplicación de correo electrónico le permite enviar y recibir mensajes de correo, y su navegador Web le permite ver páginas de sitios Web en todo el mundo. Los programadores de computadoras crean dichas aplicaciones, escribiendo programas de cómputo que permiten a los usuarios de computadoras llevar a cabo sus tareas diarias.

Consideremos una *aplicación* sencilla que muestra una línea de texto. Una aplicación en Java es un programa que se ejecuta utilizando el intérprete `java`. (Más adelante en esta sección hablaremos sobre cómo compilar y ejecutar un programa.) El programa y su salida se muestran en la figura 2.1. El programa ilustra varias características importantes del lenguaje Java. Java utiliza notaciones que pueden parecer extrañas a los no programadores. Además, cada uno de los programas que presentamos en este libro tiene números de línea incluidos para su conveniencia; los números de línea no son parte de los programas en Java. Pronto veremos que la línea 9 se encarga del "verdadero trabajo" del programa; a saber, mostrar la frase `¡Bienvenido a la programación en Java!` en la pantalla. Ahora consideremos cada línea del programa en orden.

La línea 1

```
// Fig. 2.1: Bienvenido1.java
```

Empieza con `//`, indicando que el resto de la línea es un *comentario*. Los programadores insertan comentarios para *documentar* los programas y mejorar la legibilidad de éstos. Los comentarios también ayudan a otras personas a leer y comprender un programa. El compilador de Java ignora estos comentarios, de manera que la computadora no hace nada cuando el programa se ejecuta. Comenzamos cada uno de los programas con un comentario, el cual indica el número de figura y el nombre del archivo.

Un comentario que comienza con `//` se llama *comentario de fin de línea* (o de *una sola línea*), ya que termina al final de la línea en la que aparece. Un comentario que se especifica con `//` puede empezar en medio de una línea, y continuar solamente hasta el final de esa línea.

http://librosysolucionarios.net


```
1  // Fig. 2.1: Bienvenido1.java
2  // Programa para imprimir texto.
3
4  public class Bienvenido1 {
5
6   // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String args[] )
8     {
9        System.out.println( "¡Bienvenido a la programación en Java!" );
10
11    } // fin del método main
12
13 } // fin de la clase Bienvenido1
```

```
¡Bienvenido a la programación en Java!
```

Figura 2.1 Programa para imprimir texto.

Los *comentarios tradicionales* (también conocidos como *comentarios de múltiples líneas*), como el que se muestra a continuación

```
/* Éste es un comentario
   tradicional. Puede
   dividirse en muchas líneas */
```

se distribuyen en varias líneas. Este tipo de comentario comienza con el delimitador `/*` y termina con `*/`. El compilador ignora todo el texto que esté entre los delimitadores. Otro tipo de comentario muy similar se conoce como *comentario Javadoc*™ y está delimitado por `/**` y `*/`.[1]



Error común de programación 2.1

Olvidar uno de los delimitadores de un comentario tradicional o Javadoc es un error de sintaxis. Este tipo de errores ocurre cuando el compilador no reconoce una línea de código del programa. El compilador generalmente muestra un mensaje de error para ayudar al programador a identificar y corregir la línea que está incorrecta. Los errores de sintaxis son violaciones a las reglas del lenguaje. Estos errores se conocen también como errores del compilador, errores en tiempo de compilación *o* errores de compilación, *ya que el compilador los detecta durante la fase de compilación. Usted no podrá ejecutar su programa sino hasta que corrija todos los errores de sintaxis que éste contenga.*

La línea 2

```
// Programa para imprimir texto.
```

es un comentario de fin de línea que describe el propósito del programa.



Buena práctica de programación 2.1

Es conveniente que todo programa comience con un comentario que explique su propósito, el autor, la fecha y la hora de su creación.[2]

1. Java incorporó los comentarios delimitados con `/*` y `*/` del lenguaje de programación C, y los comentarios de fin de línea, delimitados con `//`, del lenguaje de programación C++. En este libro utilizamos comentarios de fin de línea, estilo C++. Los comentarios delimitados por `/**` y `*/` son casos especiales de comentarios tradicionales. Estos comentarios permiten a los programadores incrustar la documentación del programa directamente en éste. Dichos comentarios son el formato preferido en la industria, en relación con el uso de comentarios en Java. El programa de utilería `javadoc` (proporcionado por Sun Microsystems con el Kit de desarrollo de software para Java2) lee esos comentarios y los utiliza para preparar la documentación de su programa, en formato HTML. Hay algunas sutilezas en cuanto al uso apropiado de los comentarios estilo Java. Nosotros no utilizamos comentarios estilo Java en los programas que se presentan en este libro, debido a las restricciones de espacio. Para obtener información completa sobre el uso de `javadoc`, visite la Página de herramientas de `javadoc` en `java.sun.com/j2se/javadoc`.
2. No mostraremos el nombre del autor, la fecha y la hora en los programas de este libro, ya que esta información sería redundante.

La línea 3 es simplemente una línea en blanco. Los programadores usan líneas en blanco y espacios para facilitar la lectura de los programas. En conjunto, las líneas en blanco, los espacios y los tabuladores se conocen como *espacio en blanco*. (Los espacios y tabuladores se conocen específicamente como *caracteres de espacio en blanco*.) El compilador ignora el espacio en blanco. En éste y en los siguientes capítulos, hablaremos sobre las convenciones para utilizar espacio en blanco para mejorar la legibilidad de los programas.



Buena práctica de programación 2.2

Utilice líneas en blanco y espacios para mejorar la legibilidad del programa.

La línea 4

```
public class Bienvenido1 {
```

comienza una *declaración de clase*[3] para la clase `Bienvenido1`. Todo programa en Java consiste de, cuando menos, una declaración de clase que usted, el programador, debe definir. Estas clases se conocen como *clases definidas por el programador* o *clases definidas por el usuario*. La *palabra clave* `class` introduce una declaración de clase en Java, la cual debe ir seguida inmediatamente por el *nombre de la clase* (`Bienvenido1`). Las palabras clave[4] (algunas veces conocidas como *palabras reservadas*) se reservan para uso exclusivo de Java (hablaremos sobre las diversas palabras clave a lo largo de este texto) y siempre se escriben en minúscula.

Por convención, todos los nombres de clases en Java comienzan con una letra mayúscula, y la primera letra de cada palabra en el nombre de la clase debe ir en mayúscula (por ejemplo, `EjemploDeNombreDeClase`). El nombre de la clase se conoce como *identificador*, y está compuesto por una serie de caracteres que pueden ser letras, dígitos, guiones bajos (_) y signos de moneda ($); sin embargo, no puede comenzar con un dígito ni tener espacios. Algunos identificadores válidos son `Bienvenido1`, `$valor`, `_valor`, `m_campoEntrada1` y `boton7`. El nombre `7boton` no es un identificador válido, ya que comienza con un dígito, y el nombre `campo entrada` tampoco lo es debido a que contiene un espacio. Java es *sensible a mayúsculas y minúsculas*; es decir, las toma como identificadores diferentes, por lo que `a1` y `A1` son distintos, pero ambos son identificadores válidos.



Buena práctica de programación 2.3

Por convención, el nombre de una clase siempre debe comenzar con una letra mayúscula, y la primera letra de cada palabra subsiguiente del nombre de la clase también debe ir en mayúscula.



Buena práctica de programación 2.4

Cuando lea un programa en Java, busque identificadores en los que la primera letra del mismo esté en mayúscula. Dichos identificadores generalmente representan clases.



Observación de ingeniería de software 2.1

Evite usar identificadores que contengan signos de moneda ($). El compilador comúnmente utiliza estos signos para crear nombres de identificadores.



Error común de programación 2.2

Java es sensible a mayúsculas y minúsculas. El no utilizar la combinación apropiada de letras minúsculas y mayúsculas para un identificador, generalmente produce un error de compilación.

En los capítulos 2 al 7, cada una de las clases que definimos comienza con la palabra clave `public`. Por el momento no describiremos su uso, simplemente la emplearemos; en el capítulo 8 se describirá detalladamente, junto con otras clases que no comienzan con esta palabra.[5]

3. Muchos programadores se refieren a esto como la *definición de una clase*. Sin embargo, la especificación del lenguaje Java (`java.sun.com/docs/books/jls`) utiliza el término "declaración de una clase".
4. La lista completa de palabras clave de Java se muestra en la figura 4.2.
5. En ocasiones anteriores, le hemos pedido que tan sólo se limite a copiar en sus propios programas ciertas características de Java que le presentamos. Hacemos esto específicamente cuando consideramos que todavía no es conveniente que usted conozca todos los detalles acerca de una característica de Java. Inicialmente, todos los programadores aprenden a programar primero lo que los otros han hecho antes que ellos. Por cada detalle que le pedimos que copie, le indicaremos en dónde se presentará la explicación detallada.

Cuando usted guarda su declaración de clase `public` en un archivo, el nombre de éste debe ser el nombre de la clase, seguido de la extensión "`.java`". Para nuestra aplicación, el nombre del archivo es `Bienvenido1.java`. Todas las declaraciones de clases en Java se guardan en archivos que terminan con la extensión "`.java`".



Error común de programación 2.3

Tanto en términos de ortografía como en ahorro de tiempo, es un error que una clase `public` tenga un nombre de archivo que no sea idéntico al nombre de la clase (más la extensión `.java`). Por lo tanto, es también un error que un archivo contenga dos o más clases `public`.



Error común de programación 2.4

Si un archivo contiene la declaración de una clase, es un error que no finalice con la extensión `.java`. Si se omite esa extensión, el compilador de Java no podrá compilar la declaración de la clase.

Una *llave izquierda* (al final de la línea 4), {, comienza el *cuerpo* de la declaración de una clase. Su correspondiente *llave derecha* (en la línea 13 de este programa), }, debe terminar la declaración de una clase. Observe que las líneas de la 6 a la 11 tienen sangría. Ésta es una de las convenciones de espaciado que se mencionaron anteriormente. Definimos cada una de las convenciones de espaciado como una *Buena práctica de programación*.



Buena práctica de programación 2.5

Siempre que escriba una llave izquierda de apertura ({) en su programa, escriba inmediatamente la llave derecha de cierre (}) y luego vuelva a colocar el cursor entre las llaves y utilice sangría para comenzar a escribir el cuerpo. Esta práctica ayuda a evitar errores debido a la omisión de una de las llaves.



Buena práctica de programación 2.6

Aplique sangría a todo el cuerpo de la declaración de cada clase, usando un "nivel" de sangría entre la llave izquierda ({) y la llave derecha (}), las cuales delimitan el cuerpo de la clase. Este formato enfatiza la estructura de la declaración de la clase, y facilita su lectura.



Buena práctica de programación 2.7

Establezca una convención para el tamaño de sangría que usted prefiera, y después aplique uniformemente esta convención. La tecla Tab *puede utilizarse para crear sangrías, pero las posiciones de los tabuladores pueden variar entre los diversos editores de texto. Le recomendamos utilizar tres espacios para formar un nivel de sangría.*



Error común de programación 2.5

Es un error de sintaxis no utilizar las llaves por pares.

La línea 5 es una línea en blanco, que se insertó para mejorar la legibilidad del programa. La línea 6

```
// el método main empieza la ejecución de la aplicación en Java
```

es un comentario de fin de línea que indica el propósito de las líneas 7 a 11 del programa.

La línea 7

```
public static void main( String args[] )
```

es el punto de inicio de toda aplicación en Java. Los paréntesis después de `main` indican que éste es un bloque de construcción del programa, al cual se le llama *método*. Las declaraciones de clases en Java generalmente contienen uno o más métodos. En una aplicación en Java, sólo uno de esos métodos debe llamarse `main` y debe definirse como se muestra en la línea 7; de no ser así, el intérprete `java` no ejecutará la aplicación. Los métodos pueden realizar tareas y devolver información una vez que las hayan concluido. La palabra clave `void` indica que este método realizará una tarea (en el caso de este programa, mostrará una línea de texto), pero no devolverá ningún tipo de información cuando complete su tarea. Más adelante veremos que muchos métodos devuelven información cuando finalizan sus tareas. Los métodos se explican detalladamente en el capítulo 6. Por ahora, simplemente copie la primera línea de `main` en sus aplicaciones en Java.[6]

6. En la línea 7, la instrucción `String args[]` que aparece entre paréntesis es una parte requerida de la declaración del método `main`. Hablaremos sobre esto en el capítulo 7, Arreglos.

http://librosysolucionarios.net

La llave izquierda ({) en la línea 8 comienza el *cuerpo de la declaración del método*.[7] Su correspondiente llave derecha (}) debe terminar el cuerpo de la declaración del método (línea 11 del programa). Observe que la línea 9 en el cuerpo del método tiene sangría entre las llaves.

Buena práctica de programación 2.8

Aplique sangría a todo el cuerpo de la declaración de cada método, usando un "nivel" de sangría entre la llave izquierda ({) y la llave derecha (}), las cuales delimitan el cuerpo del método. Este formato resalta la estructura del método y ayuda a facilitar la declaración del mismo.

La línea 9

```
System.out.println( "¡Bienvenido a la programación en Java!" );
```

indica a la computadora que *realice una acción*; es decir, que imprima la *cadena* de caracteres contenida entre los caracteres de comillas dobles. A una cadena también se le denomina *cadena de caracteres*, *mensaje* o *literal de cadena*. Genéricamente, nos referimos a los caracteres entre comillas dobles como *cadenas*. El compilador no ignora los caracteres de espacio en blanco dentro de las cadenas.

Error común de programación 2.6

Es un error de sintaxis que una cadena no aparezca entre caracteres de comillas dobles, en una línea de un programa.

`System.out` se conoce como el *objeto de salida estándar*. `System.out` permite a las aplicaciones en Java mostrar conjuntos de caracteres en la *ventana de comandos*, desde la cual se ejecuta la aplicación en Java. En Microsoft Windows 95/98/ME, la ventana de comandos es el *símbolo de MS-DOS*. En Microsoft Windows NT/2000/XP, la ventana de comandos es el *Símbolo del sistema*. En UNIX/Linux/Mac OS X, la ventana de comandos se llama *ventana de terminal* o *shell*.

El método `System.out.println` *muestra* (o *imprime*) *una línea* de texto en la ventana de comandos. La cadena dentro de los paréntesis en la línea 9 es el *argumento* para el método. El método `System.out.println` realiza su tarea, mostrando su argumento en la ventana de comandos. Cuando `System.out.println` completa su tarea, posiciona el *cursor de salida* (la ubicación en donde se mostrará el siguiente carácter en la ventana de comandos) al principio de la siguiente línea en la ventana de comandos. [Este desplazamiento del cursor es similar a cuando un usuario oprime la tecla *Intro*, al escribir en un editor de texto (el cursor aparece al principio de la siguiente línea en su archivo).]

Toda la línea 9, incluyendo `System.out.println`, el argumento `"¡Bienvenido a la programación en Java!"` entre paréntesis y el *punto y coma* (`;`), se conoce como una *instrucción*. La mayoría de las instrucciones terminan con un punto y coma. Cuando se ejecuta la instrucción de la línea 9 de nuestro programa, ésta muestra el mensaje `¡Bienvenido a la programación en Java!` en la ventana de comandos. Por lo general, un método está compuesto de una o más instrucciones que realizan la tarea de ese método, como veremos en los siguientes programas.

Error común de programación 2.7

Omitir el punto y coma al final de una instrucción es un error de sintaxis.

Tip para prevenir errores 2.1

Al aprender a programar, es conveniente "descomponer" un programa funcional, para que de esta manera pueda familiarizarse con los mensajes de error de sintaxis del compilador. Estos mensajes no siempre indican el problema exacto en el código. Cuando se encuentre dichos mensajes de error de sintaxis en el futuro, tendrá una idea de qué fue lo que ocasionó el error. Trate de quitar un punto y coma o una llave del programa de la figura 2.1, y vuelva a compilarlo de manera que pueda ver los mensajes de error generados por esta omisión.

Tip para prevenir errores 2.2

Cuando el compilador reporta un error de sintaxis, éste tal vez no se encuentre en el número de línea indicado por el mensaje de error. Primero verifique la línea en la que se reportó el error; en caso de no encontrarlo ahí, verifique las líneas anteriores en el programa.

7. Muchos programadores se refieren a esto como la *definición de un método*. Sin embargo, la *especificación del lenguaje Java* (`java.sun.com/docs/books/jls`) utiliza el término "declaración de un método".

http://librosysolucionarios.net

A algunos programadores se les dificulta, cuando leen o escriben un programa, relacionar las llaves izquierda y derecha ({ y }) que delimitan el cuerpo de la declaración de una clase o de un método. Por esta razón, incluyen un comentario de fin de línea después de una llave derecha de cierre (}) que termina la declaración de un método y que termina la declaración de una clase. Por ejemplo, la línea 11

```
} // fin del método main
```

especifica la llave derecha de cierre (}) del método `main`, y la línea 13

```
} // fin de la clase Bienvenido1
```

especifica la llave derecha de cierre (}) de la clase `Bienvenido1`. Cada comentario indica el método o la clase que termina con esa llave derecha. Nosotros utilizamos dichos comentarios para ayudar a los programadores principiantes a que determinen en dónde termina el componente de cada programa. Después del capítulo 6 utilizamos dichos comentarios cuando los pares de llaves contienen muchas instrucciones, lo cual hace que las llaves de cierre sean difíciles de identificar.

Buena práctica de programación 2.9

Algo que mejora la legibilidad de los programas es añadir un comentario de fin de línea, que indique a cuál declaración de método o clase pertenece esa llave, después de la llave derecha de cierre (}) en el cuerpo de un método, o la declaración de una clase.

Cómo compilar y ejecutar su primera aplicación en Java[8]

Ahora estamos listos para compilar y ejecutar nuestro programa. Para compilar el programa, abra una ventana de comandos, cambie al directorio en donde está guardado el programa y escriba

```
javac Bienvenido1.java
```

Si el programa no contiene errores de sintaxis, el comando anterior crea un nuevo archivo llamado `Bienvenido1.class` (conocido como el *archivo de clase* para `Bienvenido1`), el cual contiene los códigos de bytes de Java que representan nuestra aplicación. Estos códigos de bytes serán interpretados por el intérprete `java` cuando le indiquemos que debe ejecutar el programa.

Tip para prevenir errores 2.3

Cuando trate de compilar un programa, si recibe un mensaje como "`comando o nombre de archivo incorrecto`". "`javac: comando no encontrado`" o "`'javac' no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable`", entonces su instalación del software Java no se completó apropiadamente. Con el Kit de desarrollo de software para Java 2, esto indica que la variable de entorno `PATH` del sistema no se estableció apropiadamente. Consulte cuidadosamente las instrucciones de instalación del Kit de desarrollo de software para Java 2. En algunos sistemas, después de corregir la variable `PATH`, es probable que necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.

Tip para prevenir errores 2.4

Cuando la sintaxis de un programa es incorrecta, el compilador de Java genera mensajes de error de sintaxis. Cada uno de estos mensajes contiene el nombre de archivo y el número de línea en donde ocurrió el error. Por ejemplo, `Bienvenido1.java:6` indica que ocurrió un error en el archivo `Bienvenido1.java` en la línea 6. El resto del mensaje de error proporciona información acerca del error de sintaxis.

La figura 2.2 muestra el programa de la figura 2.1 ejecutándose en una ventana de comandos de Microsoft Windows 2000. Para ejecutar el programa, escribimos `java Bienvenido1`, con lo que se inicia el intérprete de `Java` y se le indica que debe cargar el archivo "`.class`" para la clase `Bienvenido1`. Observe que la extensión "`.class`" del nombre de archivo se omite en el comando anterior; de no ser así, el intérprete no

8. En nuestro sitio Web, en la página `www.deitel.com/books/downloads.html`, tenemos las publicaciones de la serie *DEITEL™ DIVE INTO™*, las cuales le ayudarán a empezar a utilizar varias herramientas populares para desarrollo con Java™, incluyendo el Kit de desarrollo de software para Java™ 2 de Sun™ Microsystems versión 1.4, Sun Microsystems SunOne Studio 4 Community Edition y Borland® JBuilder™ 7 Personal. Pondremos a disposición del público más publicaciones de la serie DIVE INTO™, a medida que los maestros las soliciten.

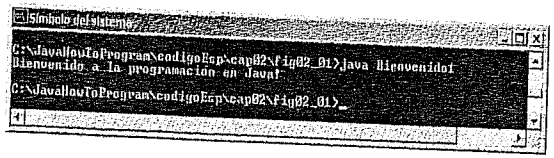


Figura 2.2 Ejecución de `Bienvenido1` en una ventana de **Símbolo del sistema** de Microsoft Windows 2000.

ejecutará el programa. El intérprete llama al método `main`. A continuación, la instrucción de la línea 9 del `main` muestra "`¡Bienvenido a la programación en Java!`"



Tip para prevenir errores 2.5

Al tratar de ejecutar un programa en Java, si recibe un mensaje como "`Exception in thread "main" java.lang.NoClassDefFoundError:Bienvenido1`*", quiere decir que su variable de entorno* `CLASSPATH` *no se ha configurado apropiadamente. Consulte cuidadosamente las instrucciones de instalación del Kit de desarrollo de software para Java 2. En algunos sistemas, tal vez necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.*

2.3 Modificación de nuestro primer programa en Java

Esta sección continúa con nuestra introducción a la programación en Java, con dos ejemplos que modifican el ejemplo de la figura 2.1 para imprimir texto en una línea utilizando varias instrucciones, y para imprimir texto en varias líneas utilizando una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones

`¡Bienvenido a la programación en Java!` puede mostrarse en varias formas. La clase `Bienvenido2`, que se muestra en la figura 2.3,[9] utiliza dos instrucciones para producir el mismo resultado que el mostrado en la figura 2.1.

El programa es casi idéntico a la figura 2.1, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Imprimir una línea de texto con varias instrucciones.
```

```
1  // Fig. 2.3: Bienvenido2.java
2  // Imprimir una línea de texto con varias instrucciones.
3
4  public class Bienvenido2 {
5
6   // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String args[] )
8     {
9        System.out.print( "¡Bienvenido a " );
10       System.out.println( "la programación en Java!" );
11
12    } // fin del método main
13
14 } // fin de la clase Bienvenido2
```

```
¡Bienvenido a la programación en Java!
```

Figura 2.3 Impresión de una línea de texto con varias instrucciones.

9. A partir de este punto en adelante, resaltaremos con pantalla las características claves y nuevas de cada programa.

es un comentario de fin de línea que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido2`.

Las líneas 9 y 10 del método `main`

```
System.out.print( "¡Bienvenido a " );
System.out.println( "la programación en Java!" );
```

muestran una línea de texto en la ventana de comandos. La primera instrucción utiliza el método `print` de `System.out` para mostrar una cadena. A diferencia de `println`, después de mostrar su argumento, `print` no posiciona el cursor de salida al inicio de la siguiente línea en la ventana de comandos; el siguiente carácter que muestra el programa en la ventana de comandos aparecerá inmediatamente después del último carácter que muestre `print`. Por lo tanto, la línea 10 coloca el primer carácter de su argumento, "l", inmediatamente después del último carácter que muestra la línea 9 (el carácter de espacio antes del carácter de comilla doble de cierre, de la cadena en la línea 9). Cada instrucción `print` o `println` continúa mostrando caracteres a partir de donde la última instrucción `print` o `println` dejó de mostrar caracteres.

Cómo mostrar varias líneas de texto con una sola instrucción

Una sola instrucción puede mostrar varias líneas, utilizando *caracteres de nueva línea*. Los caracteres de nueva línea son caracteres especiales que indican a los métodos `print` y `println` de `System.out` cuándo deben colocar el cursor de salida al inicio de la siguiente línea en la ventana de comandos. Al igual que las líneas en blanco, los espacios y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. La figura 2.4 muestra cuatro líneas de texto, utilizando caracteres de nueva línea para determinar cuándo empezar cada nueva línea.

El programa es casi idéntico al de las figuras 2.1 y 2.3, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Imprimir varias líneas de texto con una sola instrucción.
```

es un comentario de fin de línea que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido3`.

La línea 9

```
System.out.println( "¡Bienvenido\na\nla programación\nen Java!" );
```

muestra cuatro líneas separadas de texto en la ventana de comandos. Por lo general, los caracteres en una cadena se muestran exactamente como aparecen en las comillas dobles. Sin embargo, observe que los dos caracte-

```
1  // Fig. 2.4: Bienvenido3.java
2  // Imprimir varias líneas de texto con una sola instrucción.
3
4  public class Bienvenido3 {
5
6   // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String args[] )
8     {
9        System.out.println( "¡Bienvenido\na\nla programación\nen Java!" );
10
11    } // fin del método main
12
13 } // fin de la clase Bienvenido3
```

```
¡Bienvenido
a
la programación
en Java!
```

Figura 2.4 Impresión de varias líneas de texto con una sola instrucción.

http://librosysolucionarios.net

res \ y n (que se repiten tres veces en la instrucción) no aparecen en la pantalla. La *barra diagonal inversa* (\) se conoce como *carácter de escape*. Este carácter indica a los métodos `print` y `println` de `System.out` que se va a imprimir un "carácter especial". Cuando aparece una barra diagonal inversa en una cadena de caracteres, Java combina el siguiente carácter con la barra diagonal inversa para formar una *secuencia de escape*. La secuencia de escape \n representa el carácter de nueva línea. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir con `System.out`, el carácter de nueva línea hace que el cursor de salida de la pantalla se desplace al inicio de la siguiente línea en la ventana de comandos. En la figura 2.5 se enlistan varias secuencias de escape comunes, con descripciones de cómo afectan la manera de mostrar caracteres en la ventana de comandos.

2.4 Cómo mostrar texto en un cuadro de diálogo

Aunque los primeros programas que se presentaron en este capítulo muestran la salida en la ventana de comandos, muchas aplicaciones en Java utilizan ventanas o *cuadros de diálogo* para mostrar la salida. Por ejemplo, los navegadores Web como Netscape o Microsoft Internet Explorer muestran las páginas Web en sus propias ventanas. Los programas de correo electrónico le permiten escribir y leer mensajes en una ventana que proporciona el mismo programa de correo. Generalmente, los cuadros de diálogo son ventanas en las que los programas muestran mensajes importantes a los usuarios del programa. La clase `JOptionPane` de Java proporciona cuadros de diálogo previamente empaquetados, los cuales permiten a los programas mostrar ventanas que contengan mensajes para los usuarios. La figura 2.6 muestra la misma cadena de la figura 2.4 en un cuadro de diálogo predefinido.

Uno de los puntos fuertes de Java es su extenso conjunto de clases predefinidas que los programadores pueden usar, en vez de tener que "reinventar la rueda". A lo largo de este libro emplearemos muchas de esas clases. Las numerosas clases predefinidas de Java se agrupan en categorías de clases relacionadas, conocidas como *paquetes*. Un paquete es una colección de clases con nombre; éstos se conocen colectivamente como la *biblioteca de clases de Java*, o la *Interfaz de Programación de Aplicaciones de Java [API (por sus siglas en inglés) de Java)]*. Los paquetes del API de Java se dividen en *básicos* y *opcionales*. Los nombres de la mayoría de los paquetes del API de Java comienzan, ya sea con "`java`" (paquetes básicos) o "`javax`" (paquetes opcionales).[10] Muchos de ellos se incluyen como parte del Kit de desarrollo de software para Java 2; en el capítulo 6 damos un vistazo general a este tema. A medida que Java continúa evolucionando, la mayoría de los nuevos paquetes se desarrollan como paquetes opcionales. Éstos a menudo pueden descargarse de `java.sun.com` y se

Secuencia de escape	Descripción
\n	Nueva línea. Coloca el cursor de la pantalla al inicio de la siguiente línea.
\t	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
\r	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea actual; no avanza a la siguiente línea. Cualquier carácter que se imprima después del retorno de carro sobrescribe los caracteres previamente impresos en esa línea.
\\	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.

\"	Doble comilla. Se usa para imprimir un carácter de doble comilla. Por ejemplo,
	`System.out.println( "\" entre comillas \"" );`
	muestra
	`"entre comillas"`

Figura 2.5 Algunas secuencias de escape comunes.

10. Algunos nombres de paquetes en el API de Java comienzan con `org`.

http://librosysolucionarios.net


```
1  // Fig. 2.6: Bienvenido4.java
2  // Imprimir varias líneas de texto en un cuadro de diálogo.
3
4  // Paquetes de Java
5  import javax.swing.JOptionPane;  // el programa usa JOptionPane
6
7  public class Bienvenido4 {
8
9   // el método main empieza la ejecución de la aplicación en Java
10    public static void main( String args[] )
11    {
12       JOptionPane.showMessageDialog(
13          null, "Bienvenido\na\nla programación\nen Java!" );
14
15       System.exit( 0 );  // terminar la aplicación con la ventana
16
17    } // fin del método main
18
19 } // fin de la clase Bienvenido4
```

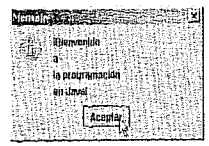


Figura 2.6 Imprimir varias líneas de texto en un cuadro de diálogo.

utilizan para mejorar las capacidades de Java. En este ejemplo utilizamos la clase predefinida de Java `JOptionPane`, la cual se encuentra en el paquete `javax.swing`.

La línea 5

```
import javax.swing.JOptionPane;  // el programa usa JOptionPane
```

es una declaración `import`. Los programadores utilizan declaraciones `import` para identificar las clases predefinidas que se utilizan en un programa en Java. El compilador trata de cerciorarse de que usted utilice correctamente las clases del API de Java. Las declaraciones `import` ayudan al compilador a localizar las clases que usted desea utilizar.[11] Por cada nueva clase que utilizamos del API de Java, debemos indicar el paquete en el que se encuentra esa clase. Esta información sobre el paquete es importante, ya que le ayuda a localizar las descripciones de cada paquete y cada clase en la *documentación del API de Java*. Puede encontrar una versión, basada en Web, de esta documentación en

```
java.sun.com/j2se/1.4/docs/api/index.html
```

También puede descargar esta documentación en su propio equipo, visitando

```
java.sun.com/docs
```

Nosotros ofrecemos un análisis general acerca del uso de esta documentación con las descargas y recursos para *Cómo programar en Java, 5a. edición* en nuestro sitio Web, `www.deitel.com`. Los paquetes se describen detalladamente en el capítulo 8, Programación basada en objetos.

11. La línea 4 de la figura 2.6 es un comentario de fin de línea, el cual indica la sección del programa en la que especificamos las declaraciones `import` para las clases en los paquetes de Java. Separamos las declaraciones `import` en los siguientes grupos: Paquetes de Java (para los nombres de paquetes que empiecen con `java` o `javax`) y paquetes Deitel (para nuestros propios paquetes que definimos más adelante en el libro).

Error común de programación 2.8

Todas las declaraciones `import` *deben aparecer antes de la declaración de la clase. Colocar una declaración* `import` *dentro del cuerpo de la declaración de una clase, o después de la declaración de una clase, es un error de sintaxis.*

La línea 5 indica al compilador que nuestro programa utiliza la clase `JOptionPane` del paquete `javax.swing`. Este paquete contiene muchas clases que ayudan a los programadores en Java a crear *Interfaces Gráficas de Usuario (GUIs, por sus siglas en inglés)* para las aplicaciones. Los *componentes de la GUI* facilitan al usuario de un programa la introducción de datos, y el formato o la presentación de los datos que se dan como resultado a ese usuario. Por ejemplo, la figura 2.7 muestra una ventana de Microsoft Internet Explorer 6. En dicha ventana, hay una barra que contiene *menús* (**Archivo**, **Edición**, **Ver**, etcétera), y por lo mismo se conoce con el nombre de *barra de menús*. Debajo de la barra de menús hay un conjunto de *botones*, cada uno de ellos especifica una tarea que Internet Explorer debe realizar, si usted hace clic en ellos. A la derecha de los botones hay un *campo de texto*, en el que puede escribir el nombre de un sitio Web que desee visitar. Los menús, botones y campos de texto forman parte de la GUI de Internet Explorer y le permiten interactuar con el programa. Java contiene clases que implementan los componentes de la GUI que se describieron anteriormente, y otros de los que hablaremos en los capítulos 13 y 14, Componentes de la interfaz gráfica de usuario: partes 1 y 2.

Tip para prevenir errores 2.6

Olvidar incluir una declaración `import` *para una clase utilizada en su programa, generalmente produce un error de compilación con el mensaje "*`cannot resolve symbol`*" (no se puede resolver el símbolo). Cuando esto ocurra, compruebe que haya proporcionado las declaraciones* `import` *correctas, y que los nombres en las declaraciones* `import` *estén escritos apropiadamente, incluyendo el uso adecuado de letras mayúsculas y minúsculas.*

En el método `main` de la figura 2.6, las líneas 12 y 13

```
JOptionPane.showMessageDialog(
   null, "¡Bienvenido\na\nla programación\nen Java!" );
```

llaman al método `showMessageDialog` de la clase `JOptionPane` para mostrar un cuadro de diálogo que contiene un mensaje. Este método requiere dos argumentos. Cuando un método requiere varios argumentos, éstos se separan con *comas* (`,`). Hasta que hablemos sobre `JOptionPane` detalladamente en el capítulo 14, el primer argumento siempre será la palabra clave `null` y el segundo argumento es la cadena a mostrar en el cuadro de diálogo. El primer argumento ayuda a la aplicación en Java a determinar en dónde se va a colocar el cuadro de diálogo. Cuando el primer argumento es `null`, el cuadro de diálogo aparece en el centro de la pantalla de la computadora.[12]

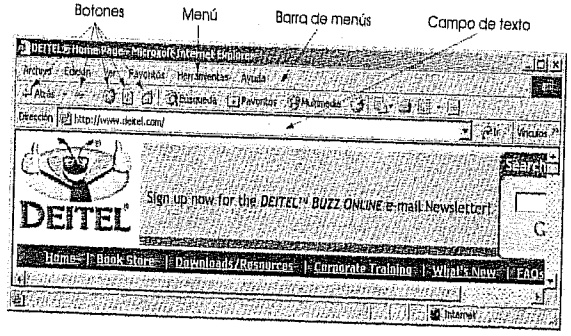


Figura 2.7 Ventana de Microsoft Internet Explorer 6 con los componentes de la GUI.

12. La mayoría de las aplicaciones que usted utiliza en su computadora se ejecutan en su propia ventana (por ejemplo, los programas de correo electrónico, navegadores Web y procesadores de palabras). Cuando dicha aplicación muestra un cuadro de diálogo, éste generalmente aparece en el centro de la ventana de aplicación, que no necesariamente se encuentra en el centro de la pantalla. Más adelante en este libro verá aplicaciones más elaboradas, en las que el primer argumento para el método `showMessageDialog` hará que el cuadro de diálogo aparezca en el centro de la ventana de aplicación, en vez de hacerlo en el centro de la pantalla.

http://librosysolucionarios.net

Buena práctica de programación 2.10

Coloque un espacio después de cada coma (`,`*) en una lista de argumentos, para que los programas sean más legibles.*

El método `JOptionPane.showMessageDialog` es un método especial de la clase `JOptionPane`, el cual se conoce como *método* `static`. Dichos métodos comúnmente definen las tareas utilizadas con frecuencia, de manera que los programadores no tengan ya que definir esas tareas por su cuenta. Por ejemplo, muchos programas muestran mensajes a los usuarios en cuadros de diálogo. En vez de requerir que los programadores creen código que realice esta tarea, los diseñadores de la clase `JOptionPane` de Java definieron un método `static` para este propósito. Ahora, con una simple llamada a un método, todos los programadores pueden hacer que aparezca un cuadro de diálogo que contenga un mensaje. Generalmente, los métodos estáticos se llaman utilizando el nombre de su clase, seguido de un punto (`.`) y el nombre del método, de la siguiente manera

NombreDeLaClase.nombreDelMetodo(argumentos)

Muchos de los métodos predefinidos que presentamos al principio de este libro son métodos `static`.[13]

Al ejecutar la instrucción de las líneas 12 y 13 se muestra el cuadro de diálogo de la figura 2.8. La *barra de título* del cuadro de diálogo contiene la cadena **Mensaje**, para indicar que este cuadro de diálogo va a presentar un mensaje al usuario. El cuadro de diálogo incluye un botón **Aceptar** (**OK**), que el usuario puede oprimir para *descartarlo* (*ocultarlo*). Esto se logra posicionando el *cursor del ratón* (también conocido como *puntero del ratón*) sobre el botón **Aceptar** y haciendo clic en el botón izquierdo del ratón (o el único botón del ratón en una Macintosh).

Recuerde que todas las instrucciones en Java terminan con un punto y coma (`;`). Por lo tanto, las líneas 12 y 13 representan una sola instrucción. Java permite que las instrucciones extensas se dividan entre muchas líneas. Sin embargo, no se puede dividir una instrucción en medio de un identificador o en medio de una cadena.

Error común de programación 2.9

Dividir una instrucción en medio de un identificador o una cadena es un error de sintaxis.

La línea 15

```
System.exit( 0 );  // terminar la aplicación con la ventana
```

utiliza el método estático `exit` de la clase `System` para terminar la aplicación. Esto se requiere para terminar cualquier aplicación que muestre una interfaz gráfica de usuario. Observe una vez más la sintaxis utilizada para llamar el método: el nombre de la clase (`System`), un punto (`.`) y el nombre del método (`exit`). Recuerde de la sección 2.2 que los identificadores que comienzan con letras mayúsculas por lo general representan nombres de clases. Por lo tanto, puede asumir que `System` es una clase. Esta clase es parte del paquete

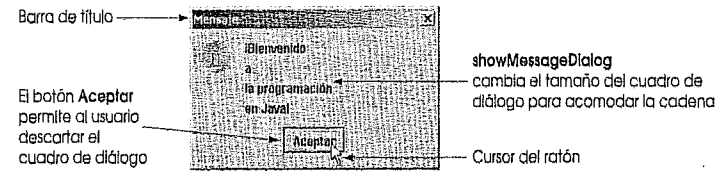


Figura 2.8 Cuadro de diálogo de mensaje.

13. Vamos a pedirle que copie nuestro uso de los métodos `static`, hasta que hablemos sobre las alternativas en el capítulo 8, Programación basada en objetos.

`java.lang`. Observe que la clase `System` no se importa con una declaración `import` al principio del programa. De manera predeterminada, el paquete `java.lang` se importa en todos los programas en Java; por lo tanto, `java.lang` es el único paquete en la API de Java que no requiere de una declaración `import`.

El argumento 0 para el método `exit` indica que el programa terminó correctamente. (Un valor distinto de cero generalmente indica que ha ocurrido un error.) Este valor se pasa a la ventana de comandos que ejecutó el programa. El argumento es útil si el programa se ejecuta desde un *archivo por lotes* (en sistemas Windows) o un *script de shell* (en sistemas UNIX/Linux/Mac OS X). Los archivos por lotes y los scripts de shell a menudo ejecutan varios programas en secuencia. Cuando termina el primer programa, el siguiente comienza a ejecutarse. Es posible utilizar el argumento para el método `exit` en un archivo por lotes o script de shell, para determinar si otro programa debería ejecutarse. Para obtener más información sobre los archivos por lotes o scripts de shell, consulte la documentación de su sistema operativo.



Error común de programación 2.10

Olvidar llamar a `System.exit` en una aplicación que muestra una interfaz gráfica de usuario evita que el programa termine correctamente. Por lo general, esta omisión ocasiona que la ventana de comandos no le permita a usted escribir más comandos. El capítulo 15 describe con más detalle la razón por la cual se requiere el uso de `System.exit` en aplicaciones basadas en GUI.

2.5 Otra aplicación en Java: Suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos *enteros* (números completos, como –22, 7 y 1024) introducidos por el usuario mediante el teclado, calcula la suma de los valores y muestra el resultado. Este programa utiliza otro cuadro de diálogo predefinido de la clase `JOptionPane`, el cual se conoce como *cuadro de diálogo de entrada* que permite al usuario introducir un valor para utilizarlo en el programa. El programa muestra un cuadro de diálogo de mensaje que contiene la suma de los enteros. Un concepto importante en este programa es que debe llevar la cuenta de los números suministrados por el usuario para poder realizar el cálculo, más adelante. Los programas llevan la cuenta de los números y demás datos en la memoria de la computadora, y utilizan estos datos mediante elementos del programa conocidos como *variables*. El programa de la figura 2.9 muestra estos conceptos.

```
1  // Fig. 2.9: Suma.java
2  // Programa que muestra la suma de dos números.
3
4  // Paquetes de Java
5  import javax.swing.JOptionPane; // el programa utiliza JOptionPane
6
7  public class Suma{
8
9     // el método main empieza la ejecución de la aplicación en Java
10    public static void main( String args[] )
11    {
12       String primerNumero; // primera cadena introducida por el usuario
13       String segundoNumero; // segunda cadena introducida por el usuario
14
15       int numero1;          // primer número a sumar
16       int numero2;          // segundo número a sumar
17       int suma;             // suma de numero1 y numero2
18
19       // leer el primer número del usuario como una cadena
20       primerNumero = JOptionPane.showInputDialog( "Escriba el primer entero" );
21
```

Figura 2.9 Programa que muestra la suma de dos números. (Parte 1 de 2.)

http://librosysolucionarios.net


```
22       // leer el segundo número del usuario como una cadena
23       segundoNumero =
24          JOptionPane.showInputDialog( "Escriba el segundo entero" );
25
26       // convertir los números de tipo String a tipo int
27       numero1 = Integer.parseInt( primerNumero );
28       numero2 = Integer.parseInt( segundoNumero );
29
30       // sumar los números
31       suma = numero1 + numero2;
32
33       // mostrar el resultado
34       JOptionPane.showMessageDialog( null, "La suma es " + suma,
35          "Resultados", JOptionPane.PLAIN_MESSAGE );
36
37       System.exit( 0 );   // terminar aplicación con la ventana
38
39    } // fin del método main
40
41 } // fin de la clase Suma
```

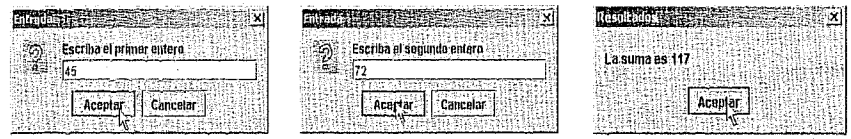


Figura 2.9 Programa que muestra la suma de dos números. (Parte 2 de 2.)

Las líneas 1 y 2

```
// Fig. 2.9: Suma.java
// Programa que muestra la suma de dos números.
```

son comentarios de fin de línea que indican el número de la figura, el nombre del archivo y el propósito del programa.

Las líneas 4 y 5

```
// Paquetes de Java
import javax.swing.JOptionPane;  // el programa utiliza JOptionPane
```

indican que el programa utiliza la clase `JOptionPane` del paquete `javax.swing`.

Como se indicó en la sección 2.2, todo programa en Java consiste de cuando menos la declaración de una clase.

La línea 7

```
public class Suma{
```

comienza la declaración de la clase `Suma`. El nombre de archivo para esta clase `public` debe ser `Suma.java`.

Recuerde que el cuerpo de la declaración de cada clase empieza con una llave izquierda de apertura { (al final de la línea 7), y termina con una llave derecha de cierre } (línea 41).

Como se dijo también en la sección 2.2, toda aplicación comienza su ejecución con el método `main` (líneas 10 a la 39). La llave izquierda (línea 11) marca el comienzo del cuerpo de `main`, y su correspondiente llave derecha (línea 39) marca el final. Observe que la declaración completa del `main` tiene un nivel de sangría en el cuerpo de la clase `Suma` y que el código tiene otro nivel de sangría, para mejorar la legibilidad.

Las líneas 12 y 13

```
String primerNumero;    // primera cadena introducida por el usuario
String segundoNumero;   // segunda cadena introducida por el usuario
```

son *instrucciones para declarar variables* (también conocidas como *declaraciones*), las cuales especifican los nombres y tipos de las *variables* que se utilizan en este programa. Una variable es una ubicación en la memoria de la computadora, en donde puede guardarse un valor para que sea usado por un programa. Todas las variables deben declararse con un *nombre* y un *tipo* antes de poder usarse. El nombre de una variable permite al programador utilizar su valor en memoria; y dicho nombre puede ser cualquier identificador válido. (Consulte en la sección 2.2 los requerimientos para nombrar identificadores.) El tipo de una variable especifica el tipo de información que se guarda en esa ubicación de memoria. Las declaraciones de las líneas 12 y 13 especifican que las variables llamadas `primerNumero` y `segundoNumero` son datos de tipo `String`; es decir, esas variables pueden almacenar secuencias de caracteres. La clase `String` se define en el paquete `java.lang`.[14] Como veremos, Java tiene muchos tipos predefinidos. Al igual que las instrucciones, las declaraciones terminan con punto y coma (`;`). Observe los comentarios de fin de línea al final de cada línea. Esta forma de usar y colocar los comentarios es una práctica común de programación que indica el propósito de cada variable en el programa.



Buena práctica de programación 2.11

Seleccionar nombres de variables significativos ayuda a que un programa se autodocumente (es decir, que sea más fácil entender el programa con sólo leerlo, en lugar de leer manuales o ver un número excesivo de comentarios).



Buena práctica de programación 2.12

Por convención, los identificadores de nombre de variables empiezan con una letra minúscula. Al igual que con los nombres de las clases, cada una de las palabras en el nombre, que van después de la primera, deben empezar con una letra mayúscula. Por ejemplo, el identificador `primerNumero` tiene una `N` mayúscula en su segunda palabra, `Numero`.



Buena práctica de programación 2.13

Declare cada variable en una línea separada. Este formato permite insertar fácilmente comentarios descriptivos en seguida de cada declaración.

Las declaraciones pueden dividirse en varias líneas, separando cada variable de la declaración con una coma (es decir, una *lista separada por comas* de nombres de variables). Varias variables del mismo tipo pueden declararse en una sola declaración, o en varias de ellas. Por ejemplo, las líneas 12 y 13 también pueden escribirse así:

```
String primerNumero,  // primera cadena introducida por el usuario
       segundoNumero; // segunda cadena introducida por el usuario
```

Las líneas 15 a 17

```
int numero1;          // primer número a sumar
int numero2;          // segundo número a sumar
int suma;             // suma de numero1 y numero2
```

declaran que las variables `numero1`, `numero2` y `suma` son datos de tipo `int`, lo cual significa que guardan valores *enteros* (número completos como 7, –11, 0 y 31,914).[15]

14. Recuerde que Java importa las clases del paquete `java.lang` por usted, de manera que no se requieren declaraciones `import` para las clases en este paquete.

15. Pronto hablaremos sobre los tipos `float` y `double`, para especificar números reales (números con puntos decimales, como `3.4`, `0.0` y `-11.19`), y las variables de tipo `char`, para especificar datos de caracteres. Una variable `char` sólo puede guardar una letra minúscula, mayúscula, un solo dígito o un solo carácter especial (por ejemplo, `x`, `A`, `7` o `*`) y secuencias de escape (como el carácter de nueva línea, `\n`). Los tipos como `int`, `double` y `char` se conocen comúnmente como *tipos primitivos*, o *tipos integrados*. Los nombres de los tipos primitivos son palabras clave; por lo tanto, deben aparecer completamente en minúsculas. El capítulo 4 sintetiza las características de los ocho tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`).

http://librosysolucionarios.net

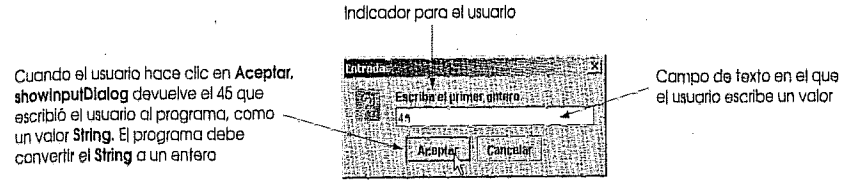


Figura 2.10 Cuadro de diálogo de entrada.

La línea 19 es un comentario de fin de línea, el cual indica que la siguiente instrucción lee el primer número del usuario. La línea 20

```
primerNumero = JOptionPane.showInputDialog( "Escriba el primer entero" );
```

utiliza el método `showInputDialog` de `JOptionPane` para mostrar el cuadro de diálogo en la figura 2.10. El argumento para `showInputDialog` indica lo que el usuario debe escribir en el campo de texto. Este mensaje se llama *indicador*, ya que hace que el usuario realice una acción específica. El usuario escribe caracteres en el campo de texto y luego hace clic en el botón **Aceptar** u oprime *Intro* para regresar la cadena al programa.[16] Desafortunadamente, la forma simple de entrada en Java que es similar a mostrar la salida en la ventana de comandos con los métodos `print` y `println` de `System.out` tiene capacidades limitadas. Por esta razón, generalmente recibimos la entrada del usuario a través de un componente de la GUI (un cuadro de diálogo de entrada en este programa).

Técnicamente el usuario puede escribir cualquier cosa en el campo de texto del cuadro de diálogo de entrada. Nuestro programa asume que el usuario seguirá las indicaciones y escribirá un valor de entero válido. En este programa, si el usuario escribe un valor no entero o hace clic en el botón **Cancelar** en el cuadro de diálogo de entrada, se producirá un error lógico en tiempo de ejecución y el programa no funcionará correctamente. El capítulo 15, Manejo de excepciones, habla sobre cómo hacer sus programas más robustos, al permitirles manejar dichos errores. Esto también se conoce como hacer que su programa sea *tolerante a fallas*.

El resultado del método `showInputDialog` de `JOptionPane` (un valor `String` que contiene los caracteres escritos por el usuario) se da a la variable `primerNumero`, utilizando el *operador de asignación*, `=`. La instrucción (línea 20) se lee como "`primerNumero` *obtiene* el valor de `JOptionPane.showInputDialog( "Escriba el primer entero" )`." El operador `=` se conoce como un *operador binario*, ya que tiene dos *operandos*: `primerNumero` y el resultado de la llamada al método `JOptionPane.showInputDialog( "Escriba el primer entero" )`. Toda esta instrucción se conoce como una *instrucción de asignación*, ya que es una instrucción que asigna un valor a una variable. Todo lo que está a la derecha del operador de asignación, `=`, siempre se evalúa primero. Las líneas 23 y 24

```
segundoNumero =
   JOptionPane.showInputDialog( "Escriba el segundo entero" );
```

muestran un cuadro de diálogo en el que el usuario escribe un valor `String` que representa al segundo de los dos enteros a sumar.

Las líneas 27 y 28

```
numero1 = Integer.parseInt( primerNumero );
numero2 = Integer.parseInt( segundoNumero );
```

convierten los dos valores `String` que introduce el usuario en valores `int`, para que el programa pueda utilizarlos en un cálculo. El método estático `parseInt` de la clase `Integer` convierte su argumento `String`

16. La primera vez que ejecute este programa, si escribe y no aparece nada en el campo de texto, coloque el puntero del ratón en el campo de texto y haga clic con el botón izquierdo del ratón (o el único botón del ratón en una Macintosh) para activar el campo de texto.

en número entero. La clase `Integer` se encuentra en el paquete `java.lang`. La línea 27 asigna el valor `int` (entero) devuelto por `parseInt`, a la variable `numero1`. La línea 28 asigna el valor `int` (entero) que `parseInt` devuelve a la variable `numero2`.

La línea 31

```
suma = numero1 + numero2;
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma`, utilizando el operador de asignación, `=`. La instrucción se lee como "`suma` *obtiene* el valor de `numero1 + numero2`." La mayoría de los cálculos se realizan en instrucciones de asignación. Cuando el programa encuentra la operación de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es un operador binario: sus dos operandos son `numero1` y `numero2`.[17]



Buena práctica de programación 2.14

Coloque espacios en cualquier lado de un operador binario, para hacer que éste destaque y haga el programa más legible.

Una vez realizado el cálculo, las líneas 34 y 35

```
JOptionPane.showMessageDialog( null, "La suma es " + suma,
   "Resultados", JOptionPane.PLAIN_MESSAGE );
```

utilizan el método estático `showMessageDialog` de la clase `JOptionPane` para mostrar el resultado de la suma. Esta nueva versión del método `showMessageDialog` de `JOptionPane` requiere cuatro argumentos. Como en la figura 2.6, el primer argumento `null` indica que el cuadro de diálogo de mensaje aparecerá en el centro de la pantalla. El segundo argumento es el mensaje a mostrar. El tercer argumento de este método en la figura 2.9 (`"Resultados"`) representa la cadena que debe aparecer en la *barra de título* del cuadro de diálogo. El cuarto argumento, `JOptionPane.PLAIN_MESSAGE`, es el *tipo de cuadro de diálogo*, un valor que indica el tipo de cuadro de diálogo de mensaje que se va a mostrar. Este tipo de cuadro de diálogo no muestra un icono a la izquierda del mensaje. La figura 2.11 muestra el segundo y tercer argumentos, y además muestra que no hay icono en la ventana (debajo de la barra de título).

En las líneas 34 y 35, el segundo argumento es la expresión

```
"La suma es " + suma
```

el cual utiliza el operador `+` para "sumar" una cadena (`"La suma es "`) y el valor de la variable `suma` (la variable `int` que contiene el resultado de la suma de la línea 31). Java tiene una versión de `+` para la *concatenación de cadenas* (combinar dos cadenas para formar una más grande). En este contexto, el operador `+` con-

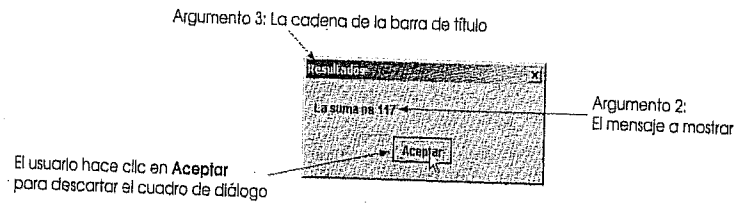


Figura 2.11 Cuadro de diálogo de mensaje personalizado con la versión de cuatro argumentos del método **showMessageDialog**.

17. Las partes de las instrucciones que contienen cálculos se llaman *expresiones*. De hecho, una expresión es cualquier parte de una instrucción que tenga un valor asociado con ella. Por ejemplo, el valor de la expresión `numero1 + numero2` es la suma de los números. De manera similar, el valor de la expresión en la línea 20 es la cadena que el usuario escribe en el cuadro de diálogo de entrada.

catena un valor `String` y un valor de otro tipo (posiblemente otro `String`); el resultado de esta operación es un nuevo (y generalmente más extenso) `String`, en el cual el operando izquierdo de `+` se encuentra al principio de la cadena y el operando derecho se encuentra al final de la cadena. Si suponemos que `suma` contiene el valor entero `117`, la expresión se evalúa de la siguiente manera:

1. Java determina que los dos operandos del operador `+` (la cadena `"La suma es "` y el entero `suma`) son de distintos tipos, y que uno de ellos es `String`,
2. Java convierte `suma` a `String` (en este caso, `"117"`).
3. Java adjunta la representación `String` de `suma` al final de `"La suma es "` (observe el espacio al final de la cadena), lo que produce la cadena `"La suma es 117"`.

A continuación, el método `showMessageDialog` muestra la cadena resultante en el cuadro de diálogo. Observe que la conversión automática del entero `suma` ocurre sólo porque el operador de suma concatena la cadena `"La suma es "` y `suma`. Además, observe que el espacio entre `es` y `117` forma parte de la cadena `"La suma es "`.

Los tipos de cuadros de diálogo de mensaje se muestran en la figura 2.12. Todos los tipos de cuadros de diálogo de mensaje, excepto `PLAIN_MESSAGE`, muestran un icono al usuario indicando el tipo del mensaje. Observe que se muestra un icono `QUESTION_MESSAGE` en un cuadro de diálogo de entrada (vea la figura 2.10).



Error común de programación 2.11

Confundir el operador `+` que se utiliza para concatenar cadenas, con el operador `+` utilizado para sumar, puede producir resultados extraños. Por ejemplo, si la variable entera `y` tiene el valor 5, la expresión `"y + 2 = " + y + 2` produce la cadena `"y + 2 = 52"`, no `"y + 2 = 7"`, ya que primero el valor de `y` se concatena con la cadena `"y + 2 = "`, y después el valor 2 se concatena con la nueva cadena más extensa `"y + 2 = 5"`. La expresión `"y + 2 = " + (y + 2)` produce el resultado deseado `"y + 2 = 7"`.

2.6 Conceptos acerca de la memoria

Los nombres de variables como `numero1`, `numero2` y `suma` en realidad corresponden a ciertas *ubicaciones* en la memoria de la computadora. Toda variable tiene un *nombre*, un *tipo*, un *tamaño* y un *valor*.

En el programa de suma de la figura 2.9, cuando la instrucción

```
numero1 = Integer.parseInt( primerNumero );
```

se ejecuta, la cadena que el usuario escribió anteriormente en el cuadro de diálogo de entrada, y que se guardó en `primerNumero`, se convierte en `int` y se coloca en una ubicación en memoria, que el compilador tenga asignada para el nombre `numero1`. Suponga que el usuario escribe la cadena `45` como el valor para `primer-`

Tipo de cuadro de diálogo de mensaje	Icono	Descripción
`JOptionPane.ERROR_MESSAGE`		Muestra un cuadro de diálogo que indica un error al usuario.
`JOptionPane.INFORMATION_MESSAGE`		Muestra un cuadro de diálogo con un mensaje informativo para el usuario. Éste puede simplemente descartar el cuadro de diálogo.
`JOptionPane.WARNING_MESSAGE`		Muestra un cuadro de diálogo que advierte al usuario de un problema potencial.
`JOptionPane.QUESTION_MESSAGE`		Muestra un cuadro de diálogo con un signo de interrogación para el usuario. Este cuadro de diálogo generalmente requiere una respuesta, como hacer clic en un botón **Sí** o **No**.
`JOptionPane.PLAIN_MESSAGE`	sin icono	Muestra un cuadro de diálogo que simplemente contiene un mensaje, sin icono.

Figura 2.12 Constantes de `JOptionPane` para los cuadros de diálogo de mensaje.

http://librosysolucionarios.net



Figura 2.13 Ubicación en memoria que muestra el nombre y el valor de la variable `numero1`.

`Numero`. El programa convierte `primerNumero` en un `int`, y la computadora coloca ese valor entero, `45`, en la ubicación `numero1`, como se muestra en la figura 2.13.

Siempre que se coloca un valor en una ubicación en memoria, este valor sustituye al valor anterior en esa ubicación. El valor anterior se pierde.

Cuando la instrucción

```
numero2 = Integer.parseInt( segundoNumero );
```

se ejecuta, suponga que el usuario escribe la cadena `72` como valor para `segundoNumero`. El programa convierte `segundoNumero` en `int`, y la computadora coloca ese valor entero, `72`, en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 2.14.

Después de que el programa de la figura 2.9 obtiene los valores para `numero1` y `numero2`, suma los valores y coloca el resultado en la variable `suma`. La instrucción

```
suma = numero1 + numero2;
```

realiza la suma y también sustituye el valor anterior de `suma`. Una vez que se ha calculado la `suma`, la memoria aparece como se muestra en la figura 2.15. Observe que los valores de `numero1` y `numero2` aparecen exactamente como lo hicieron antes de usarse en el cálculo de `suma`. Estos valores se usaron, pero no se destruyeron, cuando la computadora realizó el cálculo. Por lo tanto, cuando se lee un valor de una ubicación en memoria, el proceso es no destructivo.

2.7 Aritmética

La mayoría de los programas realizan cálculos aritméticos. Los *operadores aritméticos* se sintetizan en la figura 2.16. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El *asterisco (*)* indica la multiplicación, y el *signo de porcentaje (%)* es el *operador residuo* (conocido como *módulo* en algunos lenguajes), el cual describiremos en breve. Los operadores aritméticos en la figura 2.16 son operadores binarios, ya que funcionan con dos operandos. Por ejemplo, la expresión `suma + valor` contiene el operador binario `+` y los dos operandos `suma` y `valor`.

La *división de enteros* produce un cociente entero: por ejemplo, la expresión `7 / 4` da como resultado `1`, y la expresión `17 / 5` da como resultado `3`. Cualquier parte fraccionaria en una división de enteros simplemente se descarta (es decir, se trunca); no ocurre un redondeo. Java proporciona el operador residuo, `%`, el cual

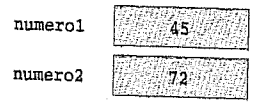


Figura 2.14 Ubicaciones en memoria después de guardar los valores para `numero1` y `numero2`.

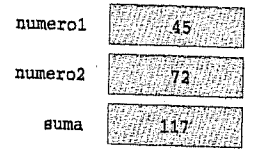


Figura 2.15 Ubicaciones en memoria después de calcular la `suma` de `numero1` y `numero2`.

http://librosysolucionarios.net

Operación en Java	Operador aritmético	Expresión algebraica	Expresión en Java
Suma	+	$f + 7$	`f + 7`
Resta	-	$p - c$	`p - c`
Multiplicación	*	bm	`b * m`
División	/	x / y o $\frac{x}{y}$ o $x \div y$	`x / y`
Residuo	%	$r \bmod s$	`r % s`

Figura 2.16 Operadores aritméticos.

produce el residuo después de la división. La expresión `x % y` produce el residuo después de que `x` se divide entre `y`. Por lo tanto, `7 % 4` produce `3`, y `17 % 5` produce `2`. Este operador se utiliza más comúnmente con operandos enteros, pero también puede usarse con otros tipos aritméticos. En capítulos posteriores, consideramos muchas aplicaciones interesantes del operador residuo, como determinar si un número es múltiplo de otro. No hay un operador aritmético para la potencia de un número en Java.[18]

Las expresiones aritméticas en Java deben escribirse en *formato de línea recta* para facilitar la escritura de programas en la computadora. Por lo tanto, las expresiones como "`a` dividida entre `b`" deben escribirse como `a / b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. La siguiente notación algebraica no es generalmente aceptable para los compiladores:

$$\frac{a}{b}$$

Los paréntesis se utilizan para agrupar términos en las expresiones en Java, de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar `a` por la cantidad `b + c`, escribimos

```
a * ( b + c )
```

Java aplica los operadores en expresiones aritméticas en una secuencia precisa, determinada por las siguientes *reglas de precedencia de operadores*, que generalmente son las mismas que las que se utilizan en álgebra:

1. Las operaciones de multiplicación, división y residuo se aplican primero. Si una expresión contiene varias operaciones de multiplicación, división o residuo, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.
2. Las operaciones de suma y resta se aplican a continuación. Si una expresión contiene varias operaciones de suma y resta, los operadores se aplican de izquierda a derecha. Los operadores de suma y resta tienen el mismo nivel de precedencia.

Las reglas de precedencia de operadores permiten a Java aplicar los operadores en el orden correcto. Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a la *asociatividad* de los operadores. Veremos que algunos operadores se asocian de derecha a izquierda. La figura 2.17 sintetiza estas reglas de precedencia de operadores. Esta tabla se expandirá a medida que se introduzcan más operadores en Java. En el apéndice A se incluye una tabla de precedencias completa.

Ahora, consideremos varias expresiones en vista de las reglas de precedencia de operadores. Cada ejemplo enlista una expresión algebraica y su equivalente en Java. El siguiente es un ejemplo de una media (promedio) aritmética de cinco términos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

Java:
```
m = ( a + b + c + d + e ) / 5;
```

18. El capítulo 5 muestra cómo realizar la potencia de un número en Java.

Operador(es)	Operación(es)	Orden de evaluación (precedencia)
*	Multiplicación	Se evalúan primero. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
/	División	
%	Residuo	
+	Suma	Se evalúan después. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
-	Resta	

Figura 2.17 Precedencia de los operadores aritméticos.

Los paréntesis son obligatorios, ya que la división tiene una mayor precedencia que la suma. La cantidad completa `( a + b + c + d + e )` va a dividirse entre `5`. Si los paréntesis se omiten por error, obtenemos `a + b + c + d + e / 5`, lo cual da como resultado

$$a + b + c + d + \frac{e}{5}$$

El siguiente es un ejemplo de la ecuación de una línea recta:

Álgebra: $y = mx + b$

Java:
```
y = m * x + b;
```

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene una mayor precedencia sobre la suma. La asignación ocurre al último, ya que tiene una menor precedencia que la multiplicación o suma.

El siguiente ejemplo contiene las operaciones residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
```
6 1 2 4 3 5

Los números dentro de los círculos bajo la instrucción, indican el orden en el que Java aplica los operadores. Las operaciones de multiplicación, residuo y división se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma y la resta. Las operaciones de suma y resta se evalúan a continuación. Estas operaciones también se aplican de izquierda a derecha.

No todas las expresiones con varios pares de paréntesis contienen paréntesis anidados. Por ejemplo, la expresión

```
a * ( b + c ) + c * ( d + e )
```

no contiene paréntesis anidados. Estos paréntesis se encuentran en el mismo nivel.

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, considere la evaluación de un polinomio de segundo grado ($y = ax^2 + bx + c$):

```
y = a * x * x + b * x + c;
```
6 1 2 4 3 5

Los números dentro de los círculos indican el orden en el que Java aplica los operadores. Las operaciones de multiplicación se evalúan primero en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma. Las operaciones de suma se evalúan a continuación. Estas operaciones también se aplican de izquierda a derecha. No existe un operador aritmético para la potencia de un número en Java, por lo que x^2 se representa como `x * x`.

Paso 1. `y = 2 * 5 * 5 + 3 * 5 + 7;`

`2 * 5 es 10` *(Multiplicación de más a la izquierda)*

Paso 2. `y = 10 * 5 + 3 * 5 + 7;`

`10 * 5 es 50` *(Multiplicación de más a la izquierda)*

Paso 3. `y = 50 + 3 * 5 + 7;`

`3 * 5 es 15` *(Multiplicación antes de la suma)*

Paso 4. `y = 50 + 15 + 7;`

`50 + 15 es 65` *(Suma de más a la izquierda)*

Paso 5. `y = 65 + 7;`

`65 + 7 es 72` *(Última suma)*

Paso 6. `y = 72;` *(Última operación; colocar 72 en y)*

Figura 2.18 Orden en el cual se evalúa un polinomio de segundo grado.

Suponga que `a`, `b`, `c` y `x` en el polinomio de segundo grado anterior se inicializan (reciben valores) como sigue: `a = 2`, `b = 3`, `c = 7` y `x = 5`. La figura 2.18 muestra el orden en el que se aplican los operadores.

Al igual que en álgebra, es aceptable colocar paréntesis innecesarios en una expresión para hacer que ésta sea más clara. A dichos paréntesis innecesarios se les llama *paréntesis redundantes*. Por ejemplo, la instrucción de asignación anterior podría colocarse entre paréntesis, de la siguiente manera:

```
y = ( a * x * x ) + ( b * x ) + c;
```



Buena práctica de programación 2.15

El uso de paréntesis para las expresiones aritméticas complejas, incluso cuando éstos no sean necesarios, puede hacer que las expresiones aritméticas sean más fáciles de leer.

2.8 Toma de decisiones: Operadores de igualdad y relacionales

Esta sección presenta una versión simple de la *instrucción* `if` de Java, la cual permite que un programa tome una decisión, dependiendo de si una *condición* es *verdadera* o *falsa*. Por ejemplo, la condición "calificación es mayor o igual que 60" determina si un estudiante pasó o no una prueba. Si la condición en una instrucción `if` es *verdadera*, el cuerpo de la instrucción `if` se ejecuta. Si la condición es *falsa*, el cuerpo no se ejecuta. Veremos un ejemplo en breve.

Las condiciones en las instrucciones `if` pueden formarse utilizando los *operadores de igualdad* y los *operadores relacionales* que se sintetizan en la figura 2.19. Ambos operadores de igualdad tienen el mismo nivel de precedencia, que es menor que la precedencia de los operadores relacionales. Los operadores de igualdad se asocian de izquierda a derecha. Los operadores relacionales tienen todos el mismo nivel de precedencia y también se asocian de izquierda a derecha.

http://librosysolucionarios.net

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores de igualdad			
=	==	x == y	x es igual a y
≠	!=	x != y	x no es igual a y
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Figura 2.19 Operadores de igualdad y relacionales.



Error común de programación 2.12

Confundir el operador de igualdad, ==, con el operador de asignación, =, puede ocasionar un error lógico o un error de sintaxis. El operador de igualdad debe leerse como "es igual a", y el operador de asignación debe leerse como "obtiene" u "obtiene el valor de". Algunas personas leen el operador de igualdad como "doble igual" o "igual igual".



Error común de programación 2.13

Es un error de sintaxis si los operadores ==, !=, >= y <= contienen espacios entre sus símbolos, como en = =, ! =, > = y < =, respectivamente.



Error común de programación 2.14

Invertir los operadores !=, >= y <=, como en =!, => y =<, es un error de sintaxis.

El siguiente ejemplo utiliza seis instrucciones `if` para comparar dos números introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es verdadera, se ejecuta la instrucción de asignación asociada. El usuario introduce dos valores a través de cuadros de diálogo de entrada. A continuación, el programa convierte los valores de entrada a enteros y los guarda en las variables **numero1** y **numero2**. Después, el programa compara los números y muestra los resultados de las comparaciones en un cuadro de diálogo de información. El programa y los resultados del ejemplo se muestran en la figura 2.20.

```
1  // Fig. 2.20: Comparación.java
2  // Compara enteros utilizando instrucciones if, operadores relacionales
3  // y de igualdad.
4
5  // Paquetes de Java
6  import javax.swing.JOptionPane;
7
8  public class Comparación {
9
10    // el método main empieza la ejecución de la aplicación en Java
11    public static void main( String args[] )
12    {
13       String primerNumero;     // primera cadena introducida por el usuario
14       String segundoNumero;    // segunda cadena introducida por el usuario
15       String resultado;        // una cadena que contiene el resultado
16
```

Figura 2.20 Operadores de igualdad y relacionales. (Parte 1 de 3.)

http://librosysolucionarios.net


```
17       int numero1;             // primer número para comparar
18       int numero2;             // segundo número para comparar
19
20       // recibir el primer número del usuario como una cadena
21       primerNumero = JOptionPane.showInputDialog( "Escriba el primer entero:" );
22
23       // recibir el segundo número del usuario como una cadena
24       segundoNumero =
25          JOptionPane.showInputDialog( "Escriba el segundo entero:" );
26
27       // convertir los números de tipo String a tipo int
28       numero1 = Integer.parseInt( primerNumero );
29       numero2 = Integer.parseInt( segundoNumero );
30
31       // inicializar resultado con cadena vacía
32       resultado = "";
33
34       if ( numero1 == numero2 )
35          resultado = resultado + numero1 + " == " + numero2;
36
37       if ( numero1 != numero2 )
38          resultado = resultado + numero1 + " != " + numero2;
39
40       if ( numero1 < numero2 )
41          resultado = resultado + "\n" + numero1 + " < " + numero2;
42
43       if ( numero1 > numero2 )
44          resultado = resultado + "\n" + numero1 + " > " + numero2;
45
46       if ( numero1 <= numero2 )
47          resultado = resultado + "\n" + numero1 + " <= " + numero2;
48
49       if ( numero1 >= numero2 )
50          resultado = resultado + "\n" + numero1 + " >= " + numero2;
51
52       // Mostrar los resultados
53       JOptionPane.showMessageDialog( null, resultado, "Resultados de la
54          comparación", JOptionPane.INFORMATION_MESSAGE );
55
56       System.exit( 0 );  // terminar la aplicación
57
58    } // fin del método main
59
60 } // fin de la clase Comparación
```

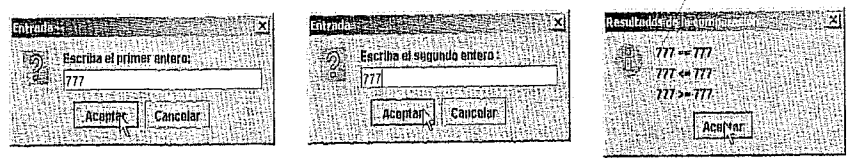


Figura 2.20 Operadores de igualdad y relacionales. (Parte 2 de 3.)

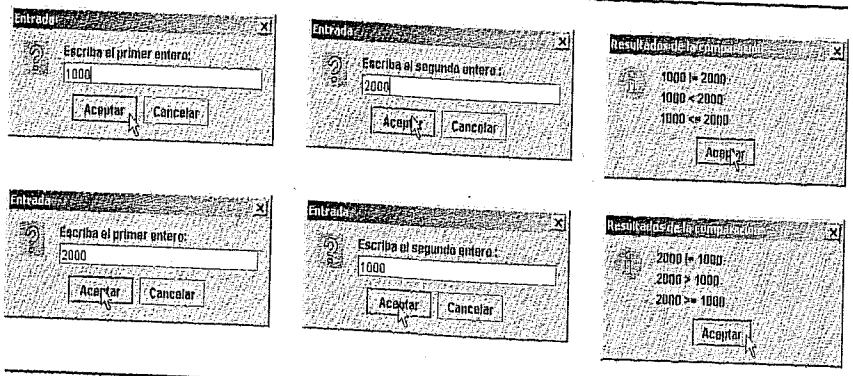


Figura 2.20 Operadores de igualdad y relacionales. (Parte 3 de 3.)

La declaración de la clase `Comparación` comienza en la línea 8

```
public class Comparación {
```

Como se describe en la sección 2.2, el método `main` (líneas 11 a 58) comienza la ejecución de toda aplicación en Java.

Las líneas 13 a 18

```
String primerNumero;      // primera cadena introducida por el usuario
String segundoNumero;     // segunda cadena introducida por el usuario
String resultado;         // una cadena que contiene el resultado

int numero1;              // primer número para comparar
int numero2;              // segundo número para comparar
```

declaran las variables que se utilizan en el método `main`. Observe que hay tres variables de tipo `String` y dos variables de tipo `int`. Recuerde de la sección 2.5 que las variables del mismo tipo pueden declararse en una o varias líneas. Una vez más, observe el comentario al final de cada declaración en las líneas 13 a 18, indicando el propósito de cada variable en el programa.

La línea 21

```
primerNumero = JOptionPane.showInputDialog( "Escriba el primer entero:" );
```

utiliza un cuadro de diálogo de entrada para permitir al usuario que escriba el primer valor entero, como una cadena, y lo guarda en `primerNumero`.

Las líneas 24 y 25

```
segundoNumero =
   JOptionPane.showInputDialog( "Escriba el segundo entero :" );
```

utilizan un cuadro de diálogo de entrada para permitir al usuario escribir el segundo valor entero, como una cadena, y lo guarda en `segundoNumero`.

Las líneas 28 y 29

```
numero1 = Integer.parseInt( primerNumero );
numero2 = Integer.parseInt( segundoNumero );
```

convierten en tipo `int` cada una de las cadenas introducidas por el usuario en los cuadros de diálogo de entrada, y asignan los valores a las variables `int` `numero1` y `numero2`.

La línea 32

```
resultado = "";
```


asigna a `resultado` la *cadena vacía* (una cadena que no contiene ningún carácter). Todas las variables que se declaran en un método (como `main`) deben inicializarse (recibir un valor) antes de que puedan usarse en una expresión. Aún no sabemos cuál será el valor final de la cadena `resultado`, por lo que vamos a asignar a esta variable la cadena vacía como valor inicial temporal.

Error común de programación 2.15

El no inicializar una variable, definida en un método, antes de utilizarla es un error de sintaxis.

Las líneas 34 y 35

```
if ( numero1 == numero2 )
   resultado = resultado + numero1 + " == " + numero2;
```

definen una *instrucción* `if` que compara los valores de las variables `numero1` y `numero2` para determinar si son iguales. Una instrucción `if` siempre comienza con la palabra clave `if`, seguida de una condición entre paréntesis, y espera una instrucción en su cuerpo. La sangría que se muestra aquí no se requiere; sin embargo, mejora la legibilidad del programa al enfatizar que la instrucción en la línea 35 es parte de la instrucción `if` que comienza en la línea 34.

Error común de programación 2.16

Olvidar los paréntesis izquierdo y derecho de la condición en una instrucción `if` es un error de sintaxis. Los paréntesis son obligatorios.

Buena práctica de programación 2.16

Aplique sangría a la instrucción en el cuerpo de una instrucción `if`, para hacer que el cuerpo destaque y para mejorar la legibilidad del programa.

Buena práctica de programación 2.17

Coloque sólo una instrucción por línea en un programa. Este formato mejora la legibilidad de los programas.

En la instrucción `if` anterior, si los valores de las variables `numero1` y `numero2` son iguales, la línea 35 asigna a `resultado` el valor de `resultado + numero1 + "==" + numero2`. Como se dijo en la figura 2.9, el operador `+` en esta expresión realiza la concatenación de cadenas. En nuestro caso, estamos suponiendo que las variables `numero1` y `numero2` tienen el valor `123`. Primero, la expresión convierte el valor de `numero1` en cadena y la anexa a `resultado` (que actualmente contiene la cadena vacía) para producir la cadena `"123"`. Luego, la expresión anexa `"=="` a `"123"` para producir la cadena `"123 == "`. Por último, la expresión anexa el valor de cadena de `numero2` a `"123 == "` para producir la cadena `"123 == 123"`. La cadena `resultado` se hace más larga, a medida que el programa avanza por las instrucciones `if` y realiza más concatenaciones. Por ejemplo, dado el valor `123` para `numero1` y `numero2` en esta discusión, las condiciones `if` en las líneas 46 (`<=`) y 49 (`>=`) también son verdaderas. Por lo tanto, el programa muestra el resultado

```
123 == 123
123 <= 123
123 >= 123
```

en un cuadro de diálogo de mensaje.

Observe que no hay punto y coma (`;`) al final de la primera línea de cada instrucción `if`. Dicho punto y coma produciría un error lógico en tiempo de ejecución. Por ejemplo, la instrucción

```
if ( numero1 == numero2 ); // error lógico
   resultado = resultado + numero1 + " ==  " + numero2;
```

sería interpretada por Java de la siguiente manera:

```
if ( numero1 == numero2 )

resultado = resultado + numero1 + " ==  " + numero2;
```

http://librosysolucionarios.net

en donde el punto y coma que aparece por sí solo en una línea (que se conoce como *instrucción vacía*) es la instrucción que se va a ejecutar si la condición en la instrucción `if` es verdadera. Al ejecutarse la instrucción vacía, no se lleva a cabo ninguna tarea en el programa. Éste continúa con la instrucción de asignación, que se ejecuta sin importar que la condición sea verdadera o falsa.



Error común de programación 2.17

Colocar un punto y coma inmediatamente después del paréntesis derecho de la condición en una instrucción `if` es, generalmente, un error lógico.

Observe el uso del espacio en blanco en la figura 2.20. Recuerde de la sección 2.3 que los caracteres de espacio en blanco, como tabuladores, nuevas líneas y espacios, generalmente son ignorados por el compilador. Por lo tanto, las instrucciones pueden dividirse en varias líneas y pueden espaciarse de acuerdo a las preferencias del programador, sin afectar el significado de un programa. Es incorrecto dividir identificadores y cadenas. Idealmente las instrucciones deben mantenerse lo más reducidas que sea posible, pero no siempre se puede hacer esto.



Buena práctica de programación 2.18

Una instrucción larga puede esparcirse en varias líneas. Si una sola instrucción debe dividirse entre varias líneas, los puntos que elija para hacer la división deben tener sentido, como después de una coma en una lista separadas por comas, o después de un operador en una expresión larga. Si una instrucción se divide entre dos o más líneas, aplique sangría a todas las líneas subsecuentes hasta el final de la instrucción.

La figura 2.21 muestra la precedencia de los operadores que se presentan en este capítulo. Los operadores se muestran de arriba abajo, en orden descendente de precedencia. Observe que todos estos operadores, con la excepción del operador de asignación, `=`, se asocian de izquierda a derecha. La suma es asociativa a la izquierda, por lo que una expresión como `x + y + z` se evalúa como si se hubiera escrito así: `( x + y ) + z`. El operador de asignación, `=`, asocia de derecha a izquierda, por lo que una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = ( y = 0 )`, en donde, como pronto veremos, primero se asigna el valor `0` a la variable `y`, y después se asigna el resultado de esa asignación, `0`, a `x`.



Buena práctica de programación 2.19

Consulte la tabla de precedencia de operadores (vea la tabla completa en el apéndice A) cuando escriba expresiones que contengan muchos operadores. Confirme que las operaciones en la expresión se realicen en el orden que usted espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzar el orden, en la misma forma que lo haría con las expresiones algebraicas. Observe que algunos operadores como el de asignación, `=`, asocian de derecha a izquierda, en vez de hacerlo de izquierda a derecha.

Hemos presentado muchas características importantes de Java en este capítulo, incluyendo: mostrar datos en la pantalla (tanto en una ventana de símbolo del sistema como en un cuadro de diálogo), introducir datos con el teclado, realizar cálculos y tomar decisiones. Las aplicaciones que se presentan aquí están diseñadas para ofrecerle una introducción a los conceptos básicos de programación. Como verá en capítulos posteriores, algunas aplicaciones en Java más substanciales contienen sólo unas cuantas líneas de código en el método `main` (esas instrucciones generalmente crean los objetos que realizan el trabajo de la aplicación). En el capítulo 3 demostraremos muchas técnicas similares, a medida que introduzcamos la programación de applets en Java. En el capítulo 4 partiremos de las técnicas de los capítulos 2 y 3, cuando presentemos la *programación estructurada*.

Operadores	Asociatividad	Tipo
`*` `/` `%`	izquierda a derecha	multiplicativa
`+` `-`	izquierda a derecha	suma
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`=`	derecha a izquierda	asignación

Figura 2.21 Precedencia y asociatividad de los operadores descritos hasta ahora.

http://librosysolucionarios.net

Usted se familiarizará más con las técnicas de sangría. Estudiaremos cómo especificar y variar el orden en el que se ejecutan las instrucciones; este orden se conoce como *flujo de control*.

2.9 (Ejemplo práctico opcional) Acerca de los objetos: Cómo examinar el enunciado de un problema

Ahora empezaremos nuestro ejemplo práctico opcional de diseño e implementación orientados a objetos. Las secciones "Acerca de los objetos" al final de este y los siguientes capítulos le ayudarán a incursionar en la orientación a objetos, mediante el análisis de un ejemplo práctico de simulación de elevador. Este ejemplo práctico le brindará una experiencia de diseño e implementación substancial, cuidadosamente pautada y completa. En los capítulos 3 al 14, 16 y 19 llevaremos a cabo los diversos pasos de un proceso de Diseño Orientado a Objetos (DOO) utilizando UML, mientras relacionamos los conceptos orientados a objetos que se describen en los capítulos. En los apéndices D, E y F implementaremos la simulación del elevador, utilizando las técnicas de la Programación Orientada a Objetos (POO) en Java. Presentaremos la solución completa al ejemplo práctico. Éste no es un ejercicio, sino una experiencia de aprendizaje de extremo a extremo, que concluye con un análisis detallado del verdadero código en Java que implementamos, con base en nuestro diseño. Hemos proporcionado este ejemplo práctico, de manera que usted pueda acostumbrarse a los tipos de problemas substanciales que se encuentran en la industria. Esperamos que disfrute esta experiencia de aprendizaje.

Enunciado del problema

Una empresa pretende construir un edificio de oficinas de dos pisos y equiparlo con un elevador. La empresa desea que usted desarrolle una *aplicación de simulador de software* que modele la operación del elevador para determinar si éste va a satisfacer las necesidades de la misma. La empresa desea que la simulación contenga un sistema de elevador consistente en un conducto y un carro elevador.

En nuestra simulación, modelamos personas que se suben al carro del elevador (al que llamaremos "el elevador") para viajar entre los pisos en el conducto del elevador, como se muestra en las figuras 2.22 a 2.24.

El elevador consiste en una puerta (a la que llamaremos la "puerta del elevador") que se abre al momento en que llega el elevador a un piso, y se cierra al momento que el elevador sale de ese piso. La puerta del elevador se mantiene cerrada durante los viajes entre los pisos, para evitar que el pasajero se lastime al rozar contra la pared del conducto del elevador. Además, el conducto se conecta a una puerta en cada piso (a las que llamaremos las dos "puertas de piso") que se cierra para que las personas no puedan caerse por el conducto cuando el elevador no se encuentre en un piso. [*Nota:* No mostramos las puertas de piso en la figuras, ya que obscurecerían la parte interior del elevador. Utilizamos una puerta de malla para representar la puerta del elevador, ya que la malla nos permite ver en el interior del elevador.] La puerta del elevador tiene un mecanismo que abre y cierra las puertas de piso cuando el elevador llega y sale, para que parezca como si ambas puertas se abrieran al mismo tiempo. Una persona sólo verá una puerta, dependiendo en qué lugar se encuentre. Una persona que esté adentro del elevador ve la puerta de éste y puede salir del elevador cuando esta puerta se abre; una persona fuera del elevador ve la puerta del piso y puede entrar al elevador cuando esa puerta se abre.[19]

El elevador empieza en el primer piso con todas las puertas cerradas. Para conservar energía, el elevador se mueve sólo cuando sea necesario. Por cuestiones de simpleza, el elevador y los pisos tienen capacidad para una persona solamente.[20]

El usuario de nuestra aplicación deberá, en todo momento, tener la habilidad de crear una persona única en la simulación, y situar a esa persona ya sea en el primer o segundo piso (figura 2.22).[21] Al ser creada, la persona camina a través del piso hacia el elevador. Si la puerta de piso no se encuentra abierta, entonces la persona oprime un botón en el piso, que está en seguida del conducto del elevador (al cual llamaremos "botón del

19. La mayoría de las personas piensan en una "puerta del elevador", cuando en realidad hay una puerta en el elevador y una puerta separada en el piso. Estas dos puertas se abren y cierran en conjunto.
20. Una vez que haya estudiado esta simulación, tal vez le sea conveniente modificarla para permitir que más de una persona suba en el elevador a la vez, y que más de una persona espere en cada piso a la vez.
21. Para crear porciones de los gráficos para la simulación del elevador, utilizamos imágenes que Microsoft proporciona en forma gratuita y que pueden descargarse en `msdn.microsoft.com/downloads/default.asp`. Creamos los demás gráficos con Paint Shop Pro™ de Jasc® Software (`www.jasc.com`).

piso"). Al oprimirlo, el botón del piso se ilumina y después solicita el elevador. Al llamarlo, el elevador viaja hacia el piso de la persona. Si el elevador se encuentra ya en el piso de esa persona, no viaja. Al momento de su llegada, el elevador restablece el botón dentro del mismo (al que llamaremos el "botón del elevador"), suena el timbre dentro del elevador y después abre la puerta del elevador (que a su vez abre la puerta del piso en ese piso). El elevador entonces indica su llegada al conducto del elevador. Éste, al recibir el mensaje, restablece el botón del piso e ilumina la luz en ese piso.

Ocasionalmente, una persona solicita el elevador cuando éste se está moviendo. Si la solicitud se generó en el piso del cual acaba de salir el elevador, éste debe "recordar" que debe volver a visitar ese piso después de llevar al pasajero actual al otro piso.

Al abrirse la puerta del piso, la persona que está esperando la llegada del elevador entra en el mismo, una vez que salga el pasajero del elevador (si hay uno). Si una persona no entra ni solicita el elevador, éste cierra su puerta y permanece en ese piso hasta que la siguiente persona oprima un botón del piso para llamar al elevador. Cuando una persona entra al elevador, esa persona oprime el botón del elevador, el cual también se ilumina al oprimirse. El elevador cierra su puerta (la cual también cierra la puerta de ese piso) y viaja al otro piso. El elevador tarda cinco segundos en viajar de un piso al otro. Cuando el elevador llega al piso de destino, la puerta del elevador se abre (junto con la puerta de ese piso) y la persona sale del elevador.

El usuario de la aplicación introduce a una persona en el primer o segundo piso oprimiendo el botón **Primer Piso** o el botón **Segundo Piso**. Cuando el usuario oprime el botón **Primer Piso**, debe crearse una persona (mediante la simulación del elevador) y colocarse en el primer piso del edificio. Cuando el usuario oprime el botón **Segundo Piso**, debe crearse una persona y colocarse en el segundo piso. Con el tiempo, el usuario puede crear cualquier número de personas en la simulación, pero no puede crear una nueva persona en un piso en el que haya otra persona esperando el elevador. Por ejemplo, la figura 2.22 muestra que el botón **Primer Piso** está deshabilitado para evitar que el usuario cree más de una persona en el primer piso. La figura 2.23 muestra que este botón se rehabilita cuando la persona entra al elevador.

La empresa solicita que mostremos la ejecución del elevador en forma gráfica, como se muestra en las figuras 2.22, 2.23 y 2.24. En puntos apropiados en el tiempo, la pantalla deberá mostrar a una persona caminar

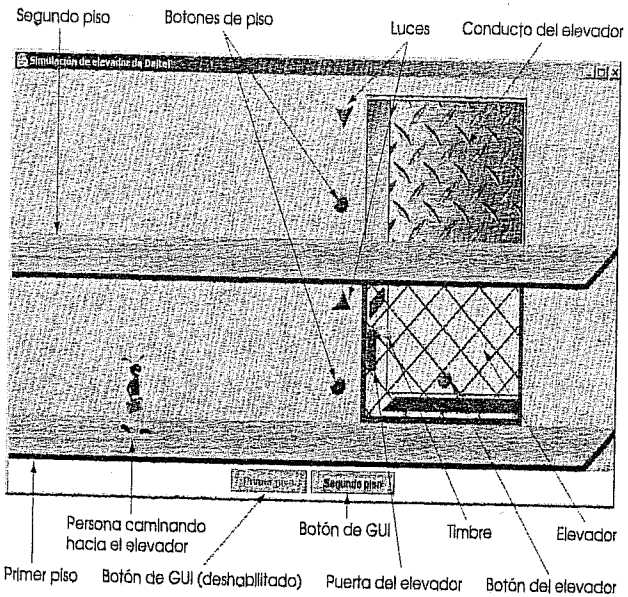


Figura 2.22 Persona avanzando hacia el elevador en el primer piso.

http://librosysolucionarios.net

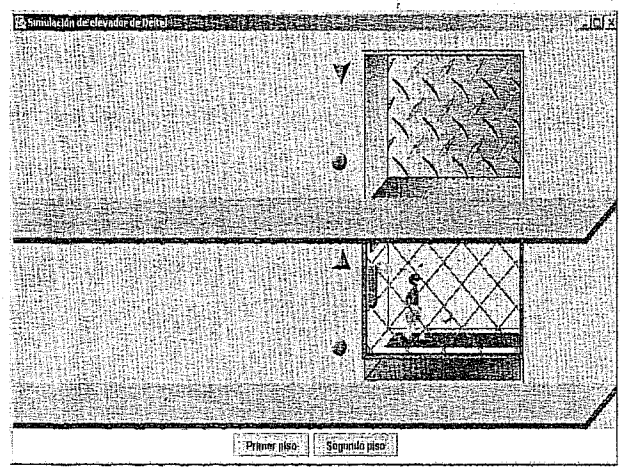


Figura 2.23 Persona subiendo al elevador que va hacia el segundo piso.

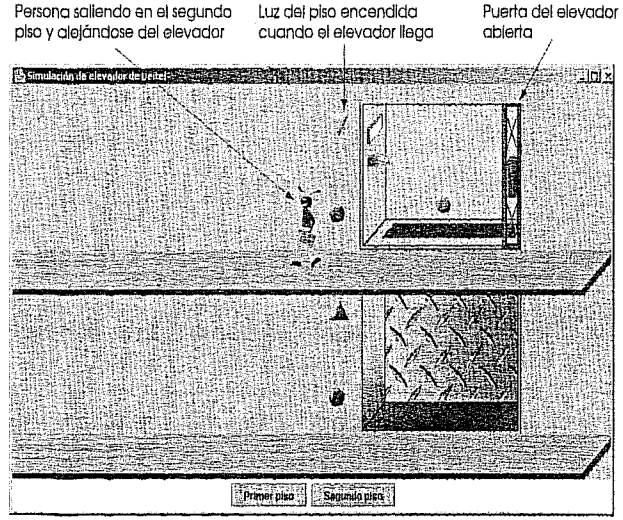


Figura 2.24 Persona alejándose del elevador.

hacia el elevador, oprimir un botón y entrar, viajar y salir del elevador. La pantalla deberá también mostrar al elevador moverse, las puertas abrirse, las luces encenderse y apagarse, los botones iluminarse al ser oprimidos y oscurecerse cuando se restablezcan.

La empresa solicita que se integre audio a la simulación. Por ejemplo, a medida que una persona vaya caminando, el usuario de la aplicación deberá escuchar los pasos. Cada vez que se oprima o restablezca un botón de piso o del elevador, el usuario debe escuchar un clic. El timbre debe sonar al momento en que llegue el elevador, las puertas deben crujir al abrir o cerrar. Por último, se debe tocar "música de elevador" a medida que el elevador viaje de un piso al otro.

Análisis y diseño del sistema del elevador

El enunciado del problema anterior que proporcionamos para este ejemplo práctico es un ejemplo simplificado de un *documento de requerimientos*; el enunciado del problema especifica el propósito general del sistema y las condiciones que debe satisfacer para resolver el problema de la compañía. Un documento de requerimientos generalmente es el resultado de un proceso detallado de *recopilación de requerimientos*, el cual podría incluir entrevistas con los usuarios del sistema y especialistas en campos específicos. Por ejemplo, un consultor que se contrata para diseñar software para un banco, podría entrevistar especialistas financieros para obtener una mejor comprensión de la manera en que el software debe funcionar.

Los arquitectos de software deben analizar el documento de requerimientos y diseñar el sistema antes de empezar la implementación del programa. Un análisis del documento de requerimientos debe producir un diseño de alto nivel para describir *qué* es lo necesario para resolver el problema. El resultado de la fase de diseño debe especificar claramente *cómo* debe construirse el sistema para satisfacer el documento de requerimientos. En las siguientes secciones "Acerca de los objetos" llevaremos a cabo los pasos de un proceso de diseño orientado a objetos (DOO) en el sistema del elevador. UML está diseñado para usarse con cualquier proceso DOO; existen muchos procesos de este tipo. Uno de los métodos populares es Rational Unified Process™, desarrollado por Rational Software Corporation. Para este ejemplo práctico presentamos nuestro propio proceso de diseño simplificado.

Ahora vamos a empezar la fase de diseño de nuestro sistema de elevador. Un sistema es un conjunto de componentes que interactúan para resolver un problema. En nuestro ejemplo práctico, la aplicación del simulador de elevador representa el sistema. Un sistema puede contener "subsistemas", los cuales son "sistemas dentro de un sistema". Los subsistemas simplifican el proceso de diseño al administrar subconjuntos de responsabilidades de un sistema. Los diseñadores del sistema deben asignar las responsabilidades del mismo entre los subsistemas, diseñar estos subsistemas y luego integrarlos con el sistema total. Hemos decidido dividir nuestro sistema de simulación de elevador en tres subsistemas:

1. la lógica del simulador (que representa la operación del sistema de elevador),
2. la representación del simulador en pantalla (de manera que el usuario pueda verlo gráficamente) y
3. la interfaz gráfica de usuario (que permite al usuario controlar la simulación).

Desarrollaremos la lógica del simulador gradualmente hasta el capítulo 16, y presentaremos la implementación en el apéndice E. En el capítulo 14 hablaremos sobre los componentes de GUI que permiten al usuario controlar el modelo, y cómo los subsistemas trabajan en conjunto para formar el sistema. Por último, introduciremos la representación del simulador en el capítulo 19 y concluiremos esta representación en el apéndice F.

La *estructura* del sistema describe sus objetos y sus relaciones. El *comportamiento* del sistema describe cómo cambia éste, a medida que sus objetos interactúan entre sí. Todos los sistemas tienen tanto estructura como comportamiento; debemos diseñar ambos. Sin embargo, existen varios *tipos* distintos de estructuras y comportamientos de los sistemas. Por ejemplo, las interacciones entre los objetos en el sistema difieren de las interacciones entre el usuario y el sistema, y aún así ambas son interacciones que constituyen una parte del comportamiento del sistema.

UML especifica nueve tipos de diagramas para modelar sistemas.[22] Cada diagrama modela una característica distinta de la estructura o comportamiento de un sistema; los primeros cuatro diagramas se relacionan con la estructura del sistema; los cinco diagramas restantes se relacionan con el comportamiento del sistema:

1. Los *diagramas de clases*, que explicaremos en la sección 3.7, modelan las clases o "bloques de construcción" que se utilizan en un sistema. Cada entidad en el enunciado del problema es candidata para ser una clase en el sistema (por ejemplo, `Persona`, `Elevador`, `Piso`, etc.).
2. Los *diagramas de objetos* modelan una "instantánea" del sistema, al modelar sus objetos y las relaciones de éstos en un punto específico en el tiempo. Cada objeto representa la instancia de una clase del diagrama de clases (por ejemplo, el objeto elevador es una instancia de la clase `Elevador`), y puede haber varios objetos creados a partir de una clase (por ejemplo, el objeto botón del primer piso y el objeto botón del segundo piso se crean a partir de la clase `BotonPiso`). El diseño para nuestra simulación de elevador no se beneficia de un diagrama de objetos, por lo que no lo incluimos en este ejemplo práctico.
3. Los *diagramas de componentes*, que presentamos en la sección 14.13, modelan los *artefactos* y *componentes* (recursos tales como gráficos, audio y archivos fuente), y los *paquetes* (que son grupos de clases) que conforman el sistema.
4. Los *diagramas de distribución* modelan los requerimientos en tiempo de ejecución del sistema (tales como la computadora o computadoras en las que residirá el sistema), los requerimientos de memoria para el sistema u otros dispositivos que éste requiera durante la ejecución. No presentamos los diagramas de distribución en este ejemplo práctico, ya que no estamos diseñando un sistema de "hardware específico"; nuestra simulación requiere sólo una computadora que contenga el entorno en tiempo de ejecución Java 2 para poder ejecutarse.
5. Los *diagramas de estados*, que presentamos en la sección 5.11, modelan *cómo* un objeto cambia de *estado* (es decir, la condición de un objeto en un tiempo específico). Cuando un objeto cambia de estado, ese objeto puede comportarse de manera distinta en el sistema.
6. Los *diagramas de actividad*, que también presentamos en la sección 5.11, modelan la *actividad* de un objeto: el flujo de trabajo del objeto durante la ejecución del programa. Un diagrama de actividad es un diagrama de flujo que modela las acciones que el objeto realizará, y en qué orden.
7. Los *diagramas de colaboración* modelan las interacciones entre objetos en un sistema, con un énfasis acerca de qué interacciones ocurren. Presentamos estos diagramas en la sección 7.10.
8. Los *diagramas de secuencia* también modelan las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de colaboración, enfatizan cuándo ocurren las interacciones. Presentamos estos diagramas en la sección 16.11.
9. Los *diagramas de caso-uso* representan la interacción entre el usuario y nuestro sistema (es decir, todas las acciones que el usuario puede realizar en el sistema). Presentamos los diagramas de caso-uso en la sección 13.17.

En la sección 3.7 continuaremos con el diseño de nuestro sistema de elevador, identificando las clases en el enunciado del problema. Lograremos esto mediante la extracción de todos los sustantivos y cláusulas nominales del enunciado del problema.[23] Mediante el uso de estas clases desarrollaremos un diagrama de clases que modele la estructura de nuestro sistema de simulación de elevador.

Recursos en Internet y World Wide Web

Los siguientes URLs proporcionan información acerca del diseño orientado a objetos con UML. Tal vez estas referencias le sean útiles a medida que estudie las secciones restantes de nuestra presentación del ejemplo práctico.

`www.omg.com/technology/uml`
Ésta es la página de recursos UML del Grupo de Administración de Objetos (OMG, por sus siglas en inglés), el cual proporciona las especificaciones para varias tecnologías orientadas a objetos, como UML.

`www.smartdraw.com/resources/centers/uml`
Este sitio muestra cómo dibujar diagramas UML sin utilizar herramientas para modelar.

`www.rational.com/uml`
Ésta es la página de recursos UML de Rational Software Corporation: la empresa que creó UML.

`microgold.com/Stage/UML_FAQ.html`
Este sitio proporciona la FAQ UML que mantiene Rational Software.

`www.softdocwiz.com/Dictionary.htm`
Este sitio contiene el Diccionario del Lenguaje de Modelado Unificado (Unified Modeling Language Dictionary), el cual enlista y define todos los términos utilizados en UML.

22. UML define tres tipos de diagramas adicionales para administrar los nueve tipos de diagramas que constituyen el modelo. No hablaremos sobre estos "diagramas de administración de modelos" (los diagramas de *Paquetes*, *Subsistemas* y *Modelos*) en este ejemplo práctico.

23. Este "análisis nominal" del enunciado del problema es parte de nuestra fase simplificada de análisis orientado a objetos (AOO). Como se mencionó anteriormente, los arquitectos de software de la industria generalmente utilizan un proceso de AOO más formal, como el prescrito por el proceso Rational Unified Process (RUP).

http://librosysolucionarios.net

`www.embarcadero.com`
Este sitio proporciona una licencia gratuita de 30 días para descargar una versión de prueba de Describe™, una herramienta para modelar con UML de Embarcadero Technologies®.

`www.togethersoft.com`
Este sitio proporciona una licencia gratuita de 30 días para descargar una versión de prueba de Together® ControlCenter™, una herramienta para desarrollar software que tiene soporte para UML.

`www.ootips.org/ood-principles.html`
Este sitio proporciona respuestas a la pregunta "¿qué necesita tener un buen DOO?"

`wdvl.internet.com/Authoring/Scripting/Tutorial/oo_design.html`
Este sitio introduce el DOO y ofrece recursos relacionados con el DOO.

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con UML. Tal vez estas referencias le sean útiles a medida que estudie las secciones restantes de nuestra presentación del ejemplo práctico.

Booch, G. *Object-Oriented Analysis and Design with Applications*. Reading, MA: Addison-Wesley, 1994.

Fowler, M. y K. Scott. *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1999.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Upper Saddle River, NJ: Prentice Hall, 1998.

Page-Jones, M. *Fundamentals of Object-Oriented Design in UML*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Software Developing Process*. Reading, MA: Addison-Wesley, 1999.

Rosenburg, D. y K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Schach, S. *Object-Oriented and Classical Software Engineering*. New York, NY: McGraw Hill, 2001.

Schneider, G. y J. Winters. *Applying Use Cases*. Reading, MA: Addison-Wesley, 1998.

Scott, K. *UML Explained*. Reading, MA: Addison-Wesley, 2001.

Stevens, P. y R. J. Pooley. *Using UML: Software Engineering with Objects and Components Revised Edition*. Reading, MA: Addison-Wesley, 2000.

RESUMEN

- Una aplicación en Java es un programa independiente que se ejecuta utilizando el intérprete `java`.
- Los programadores insertan comentarios para documentar los programas y mejorar su legibilidad. Un comentario que empieza con `//` es un comentario de fin de línea. Un comentario que comienza con `/*` y termina con `*/` es un comentario tradicional y puede abarcar varias líneas.
- Una cadena de caracteres contenidos entre comillas dobles se conoce como cadena, cadena de caracteres, mensaje o literal de cadena.
- Las líneas en blanco, los caracteres de espacio, nueva línea y tabuladores son caracteres de espacio en blanco, que, cuando se encuentran fuera de las cadenas, son ignorados por el compilador.
- Las palabras clave están reservadas para su uso en Java y deben aparecer completamente en minúsculas.
- La palabra clave `class` introduce la declaración de una clase y va seguida inmediatamente por el nombre de la clase que, por convención debe comenzar con mayúscula. Si el nombre de una clase contiene más de una palabra, la primera letra de cada palabra debe estar en mayúscula.
- Un identificador es una serie de caracteres que consisten en letras, dígitos, guiones bajos (_) y signos de moneda ($) que no empieza con un dígito, no contiene espacios y no es una palabra clave.
- Java es sensible a mayúsculas y minúsculas; es decir, distingue la diferencia entre letras mayúsculas y minúsculas.
- Una llave izquierda, `{`, y una derecha, `}`, delimitan el cuerpo de la declaración de una clase.

- Los métodos pueden realizar tareas y devolver información cuando las han complementado.
- Las aplicaciones en Java comienzan su ejecución en el método `main`. La primera línea del método `main` debe ser

```
public static void main( String args[] )
```

- Una llave izquierda, `{`, y una derecha, `}`, delimitan el cuerpo de la declaración de un método.
- `System.out` es el objeto de salida estándar. `System.out` permite a las aplicaciones en Java mostrar cadenas en la ventana de comandos en la que se esté ejecutando la aplicación.
- Los métodos `print` y `println` del objeto `System.out` muestran información en la ventana de comandos. Cuando el método `print` termina de mostrar su argumento, no coloca el cursor de salida al principio de la siguiente línea en la ventana de comandos. El siguiente carácter mostrado en la ventana de comandos aparece inmediatamente después del último carácter mostrado con `print`. Cuando `println` completa su tarea, coloca el cursor de salida al principio de la siguiente línea en la ventana de comandos.
- La secuencia de escape `\n` indica un carácter de nueva línea. Otras secuencias de escape son `\t` (tabulador), `\r` (retorno de carro), `\\` (barra diagonal inversa) y `\"` (comilla doble).
- Cada instrucción debe terminar con un punto y coma (el terminador de la instrucción).
- Java contiene muchas clases predefinidas que se agrupan en paquetes (categorías de clases relacionadas). En conjunto, estos paquetes forman la biblioteca de clases de Java, conocida como API de Java.
- El paquete `javax.swing` contiene muchas clases para crear la Interfaz Gráfica de Usuario (GUI) de una aplicación. Los componentes de GUI facilitan la entrada del usuario y la salida del programa.
- La clase `JOptionPane` se define en el paquete `javax.swing`. La clase `JOptionPane` contiene métodos que muestran cuadros de diálogo.
- Los programadores utilizan declaraciones `import` para especificar las clases requeridas para compilar un programa.
- El método `showMessageDialog` de `JOptionPane` muestra un cuadro de diálogo que contiene un mensaje.
- Para llamar a un método `static` se utiliza el nombre de su clase, seguido de un punto (`.`) y el nombre del método.
- El método `exit` de la clase `System` termina una aplicación. La clase `System` se encuentra en el paquete `java.lang`, que todos los programas en Java importan de manera predeterminada.
- Una variable es una ubicación en la memoria de la computadora, en donde puede guardarse un valor para que un programa lo utilice. Todas las variables tienen un nombre, tipo, tamaño y valor. El nombre de una variable puede ser cualquier identificador válido. Todas las variables deben declararse con un nombre y un tipo para que puedan utilizarse en un programa.
- Las declaraciones terminan con un punto y coma (`;`) y pueden dividirse en varias líneas, separando cada variable en la declaración con una coma (formando una lista, separada por comas, de nombres de variables).
- Las variables de tipo `int` guardan valores enteros.
- Los tipos como `int`, `float`, `double` y `char` son tipos primitivos; y sus nombres son palabras clave del lenguaje de programación Java.
- El "prompt" indica al usuario a que realice una acción específica.
- A una variable se le asigna un valor mediante una instrucción de asignación, la cual utiliza el operador de asignación, `=`. A este operador se le llama operador binario, ya que tiene dos operandos.
- Cuando se coloca un valor en una ubicación de memoria, el nuevo valor sustituye al valor que se encontraba anteriormente en esa ubicación. Cuando se lee un valor de una ubicación en memoria, el valor de la variable permanece sin cambio.
- El método `parseInt` de la clase `Integer` convierte su argumento `String` en un valor `int`.
- Java utiliza el operador `+` en la concatenación de cadenas, para permitir la concatenación de una cadena y un valor de otro tipo (posiblemente otra cadena).
- Los operadores aritméticos son operadores binarios, ya que cada uno de ellos opera sobre dos operandos.
- La división de enteros produce un resultado entero; se descarta la parte fraccional del resultado, en caso de haberla.
- Las expresiones aritméticas en Java deben escribirse en formato de línea recta.
- Los operadores en las expresiones aritméticas se aplican en una secuencia precisa, determinada por las reglas de precedencia de los operadores. Pueden utilizarse paréntesis para agrupar expresiones.
- Algunos operadores se asocian de izquierda a derecha; otros se asocian de derecha a izquierda.
- La instrucción `if` de Java permite a un programa tomar una decisión, dependiendo de si una condición es verdadera o falsa. Si la condición es verdadera, se ejecuta el cuerpo de la instrucción `if`. Si es falsa, no se ejecuta.
- Las condiciones en las instrucciones `if` pueden formarse utilizando los operadores de igualdad y relacionales.
- Una cadena vacía es una cadena que no contiene caracteres.
- Toda variable declarada en un método debe inicializarse para poder utilizarse.

http://librosysolucionarios.net

TERMINOLOGÍA

aplicación
argumento para un método
asociatividad de derecha a izquierda
asociatividad de los operadores
barra de título de un cuadro de diálogo
barra diagonal inversa (\), carácter de escape
Biblioteca de clases de Java
cadena
cadena de caracteres
cadena vacía ("")
carácter de nueva línea (\n)
clase
clase definida por el programador
clase definida por el usuario
`.class`, extensión de archivo
`class`, palabra clave
comentario
comentario de documentación (`/** */`)
comentario de fin de línea (`//`)
comentario tradicional (`/* */`)
compilador
concatenación de cadenas
condición
cuadro de diálogo
cuadro de diálogo de entrada
cuadro de diálogo de mensaje
cuerpo de la declaración de un método
cuerpo de la declaración de una clase
cursor del ratón
decisión
declaración
declaración de una clase
declaración de una variable
declaración `import`
división de enteros
documentar un programa
error de compilación
error de compilador
error de sintaxis
error en tiempo de compilación
espacio en blanco
`exit`, método de la clase `System`
falso
formato de línea recta
identificador
indicador
instrucción
instrucción de asignación
instrucción `if`
`int` (integer), tipo primitivo
`Integer`, clase
Interfaz de Programación de Aplicaciones (APIs) de Java
Interfaz Gráfica de Usuario (GUI)
intérprete
`.java`, extensión de archivo
`java`, intérprete
`java.lang`, paquete
Javadoc, comentario
`javax.swing`, paquete
`JOptionPane`, clase
`JOptionPane.ERROR_MESSAGE`
`JOptionPane.INFORMATION_MESSAGE`
`JOptionPane.PLAIN_MESSAGE`
`JOptionPane.QUESTION_MESSAGE`
`JOptionPane.WARNING_MESSAGE`
lista separada por comas
literal
llaves ({ y })
`main`, método
memoria
mensaje
método
nombre de una clase
nombre de una variable
objeto
objeto de salida estándar
operador
operador binario
operador de asignación (=)
operador de concatenación de cadenas (+)
operador de división (/)
operador de multiplicación (*)
operador de resta (-)
operador de suma (+)
operador residuo (%)
operadores aritméticos (*, /, %, + y -)
operadores de igualdad
  == "es igual a"
  != "no es igual a"
operadores relacionales
  >, "es mayor que"
  >=, "es mayor o igual que"
  <, "es menor que"
  <=, "es menor o igual que"
operando
palabras reservadas
paquete
paréntesis anidados
paréntesis, ()
`parseInt`, método de la clase `Integer`
precedencia
puntero del ratón
punto y coma (;), terminador de instrucción
reglas de precedencia de operadores
secuencia de escape
sensible a mayúsculas y minúsculas
`showInputDialog`, método de `JOptionPane`
`showMessageDialog`, método de la clase `JOptionPane`
`static`, método
`String`, clase
`System`, clase
`System.out`, objeto
`System.out.print`, método
`System.out.println`, método
terminador de instrucción (;)
tipo
tipo primitivo
ubicación en memoria
valor de una variable
variable
ventana de terminal
verdadero
`void`, palabra clave

EJERCICIOS DE AUTOEVALUACIÓN

2.1 Complete las siguientes oraciones:

a) El cuerpo de cualquier método comienza con un(a) _____ y termina con un(a) _____.
b) Toda instrucción termina con un _____.
c) La instrucción _____ (presentada en este capítulo) se utiliza para tomar decisiones.
d) _____ indica el inicio de un comentario de fin de línea.
e) _____, _____, _____ y _____ se conocen como espacio en blanco.
f) La clase _____ contiene métodos que muestran cuadros de diálogo de mensaje y de entrada.
g) Las _____ están reservadas para su uso en Java.
h) Las aplicaciones en Java comienzan su ejecución en el método _____.
i) Los métodos _____ y _____ muestran información en la ventana de comandos.
j) Para llamar a un método _____, se utiliza el nombre de su clase, seguido de un punto (`.`) y el nombre de ese método.

2.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Los comentarios hacen que la computadora imprima el texto que va después de los caracteres // en la pantalla, al ejecutarse el programa.
b) Todas las variables deben recibir un tipo cuando se declaran.
c) Java considera que las variables `numero` y `NuMeRo` son idénticas.
d) El operador residuo (%) puede utilizarse solamente con operandos enteros.
e) Los operadores aritméticos *, /, %, + y - tienen todos el mismo nivel de precedencia.
f) El método `Integer.parseInt` convierte un entero en `String`.

2.3 Escriba declaraciones o instrucciones en Java para realizar cada una de las siguientes tareas:

a) Declarar las variables `c`, `estaEsUnaVariable`, `q76354` y `numero` como de tipo `int`.
b) Mostrar un cuadro de diálogo que pida al usuario que escriba un entero y asigne el resultado a la variable `valor` de tipo `String`.
c) Convertir el valor `String` del inciso (b) en un entero, y guardar el valor convertido en la variable entera `edad`.
d) Si la variable `número` no es igual a 7, mostrar `"La variable número no es igual a 7"` en un cuadro de diálogo de mensaje. Utilice la versión del cuadro de diálogo de mensaje que requiere dos argumentos.
e) Imprimir `"Éste es un programa en Java"` en una línea de la ventana de comandos.
f) Imprimir `"Éste es un programa en Java"` en dos líneas de la ventana de comandos; la segunda línea debe terminar con `Java`. Utilice sólo una instrucción.

2.4 Identifique y corrija los errores en cada una de las siguientes instrucciones:

a)
```
if ( c < 7 );
   JOptionPane.showMessageDialog( null, "c es menor que 7" );
```
b)
```
if ( c => 7 )
   JOptionPane.showMessageDialog( null,
"c es igual o mayor que 7" );
```

2.5 Escriba declaraciones, instrucciones o comentarios para realizar cada una de las siguientes tareas:

a) Indicar que un programa calculará el producto de tres enteros.
b) Declarar las variables `x`, `y`, `z` y `resultado` de tipo `int`.
c) Declarar las variables `xVal`, `yVal` y `zVal` de tipo `String`.
d) Pedir al usuario que escriba el primer valor, leer el valor del usuario y guardarlo en la variable `xVal`.
e) Pedir al usuario que escriba el segundo valor, leer el valor del usuario y guardarlo en la variable `yVal`.
f) Pedir al usuario que escriba el tercer valor, leer el valor del usuario y guardarlo en la variable `zVal`.
g) Convertir `xVal` en `int` y guardar el resultado en la variable `x`.
h) Convertir `yVal` en `int` y guardar el resultado en la variable `y`.
i) Convertir `zVal` en `int` y guardar el resultado en la variable `z`.
j) Calcular el producto de los tres enteros contenidos en las variables `x`, `y` y `z`, y asignar el resultado a la variable `resultado`.
k) Mostrar un cuadro de diálogo con el mensaje `"El producto es "`, seguido del valor de la variable `resultado`.
l) Terminar el programa e indicar una terminación correcta.

2.6 Utilizando las instrucciones que escribió en el ejercicio 2.5, escriba un programa completo que calcule e imprima el producto de tres enteros.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

2.1 a) Llave izquierda ({), llave derecha (}). b) Punto y coma (;). c) `if`. d) `//`. e) Líneas en blanco, caracteres de espacio, caracteres de nueva línea y caracteres de tabulación. f) `JOptionPane`. g) Palabras clave. h) `main`. i) `System.out.print` y `System.out.println`. j) `static`.

2.2 a) Falso. Los comentarios no producen ninguna acción cuando el programa se ejecuta. Se utilizan para documentar programas y mejorar su legibilidad.

http://librosysolucionarios.net

b) Verdadero.
c) Falso. Java es sensible a mayúsculas y minúsculas, por lo que estas variables son distintas.
d) Falso. El operador residuo puede utilizarse también con operandos no enteros en Java.
e) Falso. Los operadores *, / y % se encuentran en el mismo nivel de precedencia, y los operadores + y - se encuentran en un nivel menor de precedencia.
f) Falso. El método `Integer.parseInt` convierte un valor `String` en un valor entero (`int`).

2.3 a)
```
int c, estaEsUnaVariable, q76354, numero;
```
o
```
int c;
int estaEsUnaVariable;
int q76354;
int numero;
```
b)
```
String valor = JOption.Pane.showInputDialog( "Escriba un entero" );
```
c)
```
int edad = Integer.parseInt( valor );
```
d)
```
if ( numero != 7 )
   JOptionPane.showMessageDialog( null,
      "La variable número no es igual a 7" );
```
e)
```
System.out.println( "Éste es un programa en Java" );
```
f)
```
System.out.println( "Éste es un programa en\nJava");
```

2.4 Las soluciones al ejercicio de autoevaluación 2.4 son las siguientes:
a) Error: Hay un punto y coma después del paréntesis derecho de la condición (`c < 7 )` en la instrucción `if`.
Corrección: Quite el punto y coma que va después del paréntesis derecho. [*Nota:* Como resultado, la instrucción de salida se ejecutará, sin importar que la condición en la instrucción `if` sea verdadera.]
b) Error: El operador relacional `=>` es incorrecto.
Corrección: Cambie `=>` a `>=`.

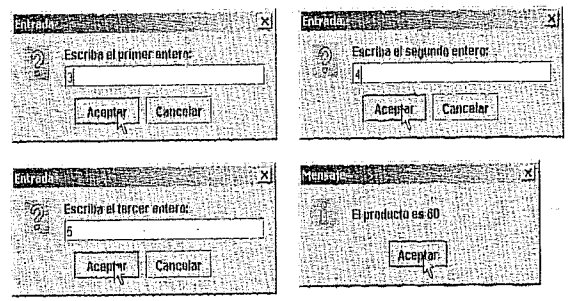
2.5 a)
```
// Calcula el producto de tres enteros
```
b)
```
int x, y, z, resultado;
```
o
```
int x;
int y;
int z;
int resultado;
```
c)
```
String xVal, yVal, zVal;
```
o
```
String xVal;
String yVal;
String zVal;
```
d)
```
xVal = JOptionPane.showInputDialog( "Escriba el primer entero:" );
```
e)
```
yVal = JOptionPane.showInputDialog( "Escriba el segundo entero:" );
```
f)
```
zVal = JOptionPane.showInputDialog( "Escriba el tercer entero:" );
```
g)
```
x = Integer.parseInt( xVal );
```
h)
```
y = Integer.parseInt( yVal );
```
i)
```
z = Integer.parseInt ( zVal );
```
j)
```
resultado = x * y * z;
```
k)
```
JOptionPane.showMessageDialog( null, "El producto es " + resultado );
```
l)
```
System.exit( 0 );
```

2.6 La solución para el ejercicio 2.6 es la siguiente:

```
1  // Ex. 2.6: Producto.java
2  // Calcular el producto de tres enteros.
3
4  // Paquetes de Java
5  import javax.swing.JOptionPane;
6
```

http://librosysolucionarios.net


```
7  public class Producto {
8
9     public static void main( String args[] )
10    {
11       int x;        // primer número
12       int y;        // segundo número
13       int z;        // tercer número
14       int resultado;  // producto de los números
15
16       String xVal;  // primera cadena introducida por el usuario
17       String yVal;  // segunda cadena introducida por el usuario
18       String zVal;  // tercera cadena introducida por el usuario
19
20       xVal = JOptionPane.showInputDialog( "Escriba el primer entero:" );
21       yVal = JOptionPane.showInputDialog( "Escriba el segundo entero:" );
22       zVal = JOptionPane.showInputDialog( "Escriba el tercer entero:" );
23
24       x = Integer.parseInt( xVal );
25       y = Integer.parseInt( yVal );
26       z = Integer.parseInt( zVal );
27
28       resultado = x * y * z;
29
30       JOptionPane.showMessageDialog( null, "El producto es " + resultado );
31
32       System.exit( 0 );
33
34    } // fin del método main
35
36 } // fin de la clase Producto
```



EJERCICIOS

2.7 Complete las siguientes oraciones:
a) _____ se utilizan para documentar un programa y mejorar su legibilidad.
b) Un cuadro de diálogo de entrada, capaz de recibir datos de entrada del usuario, se muestra con el método _____ de la clase _____.
c) Una decisión puede tomarse en un programa en Java con un(a) _____.
d) Los cálculos se realizan normalmente mediante instrucciones _____.
e) Un cuadro de diálogo capaz de mostrar un mensaje al usuario se muestra con el método _____ de la clase _____.

2.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:
a) Mostrar el mensaje "`Escriba dos números`", utilizando la clase `JOptionPane`.
b) Asignar el producto de las variables `b` y `c` a la variable `a`.
c) Indicar que un programa va a realizar un cálculo de nómina de muestra (es decir, usar texto que ayude a documentar un programa).

2.9 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Los operadores en Java se evalúan de izquierda a derecha.
b) Los siguientes nombres de variables son todos válidos: `_barra_inferior_`, `m928134`, `t5`, `j7`, `sus_ventas$`, `su_$cuenta_total`, `a`, `b$`, `C`, `z` y `z2`.
c) Una expresión aritmética válida en Java sin paréntesis se evalúa de izquierda a derecha.
d) Los siguientes nombres de variables son todos inválidos: `3g`, `87`, `67h2`, `h22` y `2h`.

2.10 Complete las siguientes oraciones:
a) Las operaciones aritméticas que tienen la misma precedencia que la multiplicación son _____.
b) Cuando los paréntesis en una expresión aritmética están anidados, ¿el conjunto _____ de paréntesis se evalúa primero?
c) Una ubicación en la memoria de la computadora, que puede contener diferentes valores en distintos momentos durante la ejecución de un programa, se conoce como _____.

2.11 ¿Qué es lo que se muestra en el cuadro de diálogo de mensaje cuando se ejecuta cada una de las siguientes instrucciones en Java? Suponga que `x = 2` y `y = 3`.
a)
```
JOptionPane.showMessageDialog( null, "x = " + x );
```
b)
```
JOptionPane.showMessageDialog( null,
   "El valor de x + x es " + ( x + x ) );
```
c)
```
JOptionPane.showMessageDialog( null, "x = " );
```
d)
```
JOptionPane.showMessageDialog(null,
   ( x + y ) + " = " + ( y + x ) );
```

2.12 ¿Cuáles de las siguientes instrucciones en Java contienen variables cuyos valores se modifican o sustituyen?
a)
```
p = i + j + k + 7;
```
b)
```
JOptionPane.showMessageDialog( null,
   "variables cuyos valores se destruyen" );
```
c)
```
JOptionPane.showMessageDialog( null, "a = 5" );
```
d)
```
cadenaVal = JOptionPane.showInputDialog( "Escriba una cadena:" );
```

2.13 Dado que $y = ax^3 + 7$, ¿cuáles de las siguientes instrucciones en Java son correctas para esta ecuación?
a)
```
y = a * x * x * x + 7;
```
b)
```
y = a * x * x * ( x + 7 );
```
c)
```
y = ( a * x ) * x * ( x + 7 );
```
d)
```
y = ( a * x ) * x * x + 7;
```
e)
```
y = a * ( x * x * x ) + 7;
```
f)
```
y = a * x * ( x * x + 7 );
```

2.14 Indique el orden de evaluación de los operadores en cada una de las siguientes instrucciones en Java, y muestre el valor de `x` después de ejecutar cada una de ellas:
a)
```
x = 7 + 3 * 6 / 2 - 1;
```
b)
```
x = 2 % 2 + 2 * 2 - 2 / 2;
```
c)
```
x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );
```

2.15 Escriba una aplicación que muestre los números del 1 al 4 en la misma línea, con cada par de números adyacentes separado por un espacio. Escriba el programa utilizando las siguientes técnicas:
a) utilizando una instrucción `System.out`.
b) utilizando cuatro instrucciones `System.out`.

2.16 Escriba una aplicación que pida al usuario que escriba dos números, que obtenga los números del usuario e imprima la suma, producto, diferencia y cociente (división) de los números. Use las técnicas que se muestran en la figura 2.9.

2.17 Escriba una aplicación que pida al usuario que escriba dos enteros, que obtenga los números del usuario y muestre el número más grande, seguido de las palabras "`es más grande`" en un cuadro de diálogo de mensaje de información. Si los números son iguales, imprima el mensaje "`Éstos números son iguales.`" Utilice las técnicas que se muestran en la figura 2.20.

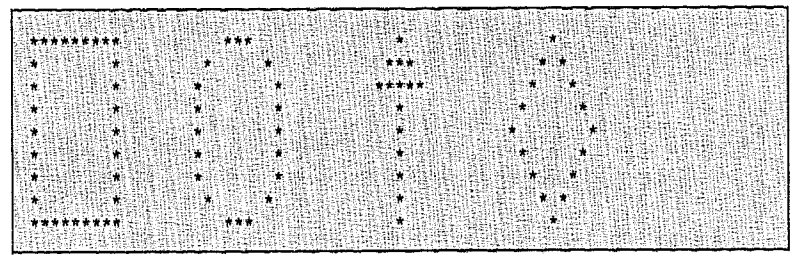
2.18 Escriba una aplicación que reciba tres enteros del usuario y muestre la suma, promedio, producto, menor y mayor de los números en un cuadro de diálogo de mensaje de información. Utilice las técnicas de GUI que se muestran en la figura 2.20. [*Nota:* El cálculo del promedio en este ejercicio debe resultar en una representación entera del promedio. Por lo tanto, si la suma de los valores es 7 el promedio debe ser 2, no 2.3333...]

2.19 Escriba una aplicación que reciba del usuario el radio de un círculo como un entero, y que imprima el diámetro, circunferencia y área de ese círculo. Utilice el valor 3.14159 para π. Utilice las técnicas de GUI que se muestran en la figura 2.9. [*Nota:* También puede utilizar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se define en el paquete `java.lang`, por lo que no es necesario importarla.] Use las siguientes fórmulas (r es el radio):

diámetro = $2r$
circunferencia = $2\pi r$
área = πr^2

No guarde los resultados de cada cálculo en una variable. En vez de eso, agregue el resultado de cada cálculo directamente a una cadena que se utilizará para mostrar los resultados.

2.20 Escriba una aplicación que muestre en la ventana de comandos un cuadro, un óvalo, una flecha y un diamante, usando asteriscos (*) como se muestra a continuación:



2.21 Modifique el programa que creó en el ejercicio 2.20 para mostrar las figuras en un cuadro de diálogo `JOptionPane.PLAIN_MESSAGE`. ¿El programa muestra las figuras exactamente igual que en el ejercicio 2.20?

2.22 ¿Qué imprime el siguiente código?
```
System.out.println( "*\n**\n***\n****\n*****" );
```

2.23 ¿Qué imprime el siguiente código?
```
System.out.println( "*" );
System.out.println( "****" );
System.out.println( "******" );
System.out.println( "*****" );
System.out.println( "***" );
```

2.24 ¿Qué imprime el siguiente código?
```
System.out.print( "*" );
System.out.print( "****" );
System.out.print( "******" );
System.out.print( "*****" );
System.out.println( "***" );
```

2.25 ¿Qué imprime el siguiente código?
```
System.out.print( "*" );
System.out.println( "****" );
System.out.println( "******" );
System.out.print( "*****" );
System.out.println( "***" );
```

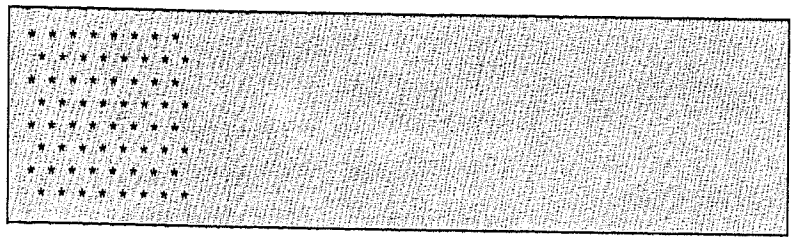
2.26 Escriba una aplicación que lea cinco enteros y que determine e imprima los enteros mayor y menor en el grupo. Use solamente las técnicas de programación que aprendió en este capítulo.

http://librosysolucionarios.net

2.27 Escriba una aplicación que lea un entero, y que determine e imprima si es par o impar. [*Sugerencia:* Use el operador residuo. Un número par es múltiplo de 2. Cualquier múltiplo de 2 deja un cociente de 0 al dividirse entre 2.]

2.28 Escriba una aplicación que lea dos enteros y determine si el primero es múltiplo del segundo y que imprima el resultado. [*Sugerencia:* Use el operador residuo.]

2.29 Escriba una aplicación que muestre en la ventana de comandos un patrón de tablero de ajedrez, como se indica a continuación:



2.30 Modifique el programa que escribió en el ejercicio 2.29 para que muestre el patrón de tablero de ajedrez en un cuadro de diálogo `JOptionPane.PLAIN_MESSAGE`. ¿El programa muestra las figuras exactamente igual que en el ejercicio 2.29?

2.31 He aquí un pequeño adelanto. En este capítulo, usted ha aprendido acerca de los enteros y el tipo `int`. Java puede también representar letras en mayúsculas, en minúsculas y una considerable variedad de símbolos especiales. Cada carácter tiene su correspondiente representación entera. El conjunto de caracteres que utiliza una computadora, y las correspondientes representaciones enteras de esos caracteres, se conocen como el *conjunto de caracteres* de esa computadora. Usted puede indicar un valor de carácter en un programa con sólo encerrar ese carácter entre comillas sencillas, como en `'A'`.

Usted puede determinar el equivalente entero de un carácter si antepone a ese carácter la palabra (`int`), como en

```
(int) 'A'
```

Esta forma se conoce como *operador de conversión de tipo*. (Hablaremos más sobre estos operadores en el capítulo 4.) La siguiente instrucción muestra un carácter y su entero equivalente:

```
System.out.println( "El carácter " + 'A' +
   " tiene el valor " + ( int ) 'A' );
```

Cuando se ejecuta esta instrucción, muestra el carácter `A` y el valor `65` (del conjunto de caracteres conocido como Unicode) como parte de la cadena.

Utilizando instrucciones similares a la mostrada anteriormente en este ejercicio, escriba una aplicación que muestre los enteros equivalentes de algunas letras en mayúsculas, en minúsculas, dígitos y símbolos especiales. Muestre los enteros equivalentes de los siguientes caracteres: `A B C a b c 0 1 2 $ * + /` y el carácter en blanco.

2.32 Escriba una aplicación que reciba del usuario un número compuesto por cinco dígitos, que separe ese número en sus dígitos individuales y los imprima, cada uno separado de los demás por tres espacios. Por ejemplo, si el usuario escribe el número `42339`, el programa debe imprimir

```
4   2   3   3   9
```

[*Sugerencia:* Es posible hacer este ejercicio con las técnicas que aprendió en este capítulo. Necesitará utilizar los operadores de división y residuo para "seleccionar" cada dígito.]

Suponga que el usuario escribe el número correcto de dígitos. ¿Qué ocurre si usted ejecuta el programa y escribe un número con más de cinco dígitos? ¿Qué ocurre si usted ejecuta el programa y escribe un número con menos de cinco dígitos?

2.33 Utilizando sólo las técnicas de programación que aprendió en este capítulo, escriba una aplicación que calcule los cuadrados y cubos de los números del 0 al 10, y que imprima los valores resultantes en formato de tabla, como se muestra a continuación. [*Nota:* Este programa no requiere de ningún tipo de entrada por parte del usuario.]


```
número  cuadrado  cubo
0       0         0
1       1         1
2       4         8
3       9         27
4       16        64
5       25        125
6       36        216
7       49        343
8       64        512
9       81        729
10      100       1000
```

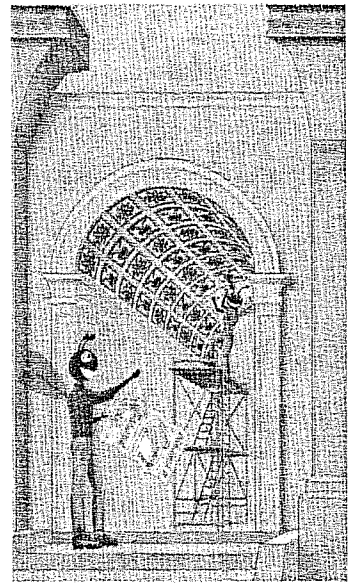
2.34 Escriba un programa que lea el primer nombre y el apellido del usuario, como dos entradas separadas, y que concatene el primer nombre y el apellido, separándolos por un espacio. Muestre el nombre concatenado en un cuadro de diálogo de mensaje.

2.35 Escriba un programa que reciba cinco números, y que determine e imprima la cantidad de números negativos, positivos, y la cantidad de ceros recibidos.

http://librosysolucionarios.net

3

Introducción a los applets de Java



Objetivos

- Diferenciar entre applets (subprogramas) y aplicaciones.
- Observar algunas de las excitantes características de Java a través de los applets de demostración incluidos en el Kit de desarrollo de software para Java 2.
- Escribir applets simples en Java.
- Escribir un documento HTML (HyperText Markup Lenguage, Lenguaje de Marcado de Hipertexto) para cargar un applet con el `appletviewer` (visor de applets) o un navegador Web, y ejecutar el applet.
- Comprender la diferencia entre variables y referencias.

Él respondería a un "¡Hola!" o a cualquier grito fuerte,
tal como "¡Mátame!" o "¡Desmenuza mi peluca!"
o "¡Como se llame!" o "¡Cualquiera que fuere su nombre!"
Lewis Caroll

La pintura es el puente que vincula la mente del pintor con la del observador.
Eugène Delacroix

Mi método es tomar el problema más grande para encontrar la frase correcta qué decir, y luego decirla con la mayor ligereza.
George Bernard Shaw

Aunque sea la locura, aún así cuenta con un método.
William Shakespeare

http://librosysolucionarios.net

Plan general

3.1 Introducción
3.2 Applets de muestra incluidos en el Kit de desarrollo de software para Java 2
3.3 Applet simple en Java: Cómo dibujar una cadena
3.4 Cómo dibujar cadenas y líneas
3.5 Cómo agregar números de punto flotante
3.6 Recursos en Internet y Web relacionados con los applets de Java
3.7 (Ejemplo práctico opcional) Acerca de los objetos: Identificación de las clases en el enunciado de un problema

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

3.1 Introducción

En el capítulo 2 presentamos la programación de aplicaciones en Java y varios aspectos importantes de éstas. También mostramos cómo ejecutar una aplicación en Java mediante el intérprete `java`. En este capítulo presentaremos otro tipo de programa en Java, conocido como *applet* (*subprograma*) de Java: un programa en Java que puede incrustarse en un *documento HTML*; es decir, una página Web. Cuando un navegador carga una página Web que contiene un applet, éste se descarga en el navegador Web y comienza a ejecutarse. Esto nos permite crear programas que cualquier usuario pueda ejecutar con sólo cargar la página Web correspondiente en su navegador Web.

El navegador que ejecuta un applet se conoce en términos genéricos como el *contenedor de applets*. El Kit de desarrollo de software para Java 2 (J2SDK) 1.4.1 incluye el contenedor de applets llamado `appletviewer` para probar los applets antes de incrustarlos en una página Web. Por lo general demostramos nuestros applets utilizando el `appletviewer`. Algunos navegadores Web utilizados actualmente no tienen soporte directo para Java 2. Netscape 7[1] es uno de los navegadores que sí ofrecen soporte para Java 2. Para ejecutar applets en otros navegadores Web como Microsoft Internet Explorer, o versiones recientes de Netscape, se requiere el *Complemento Java (Java Plug-in)*, el cual se instala como parte del J2SDK 1.4.1.

Para obtener un refuerzo de los conceptos anteriores, este capítulo repasa varios de los temas presentados en el capítulo 2. En este capítulo también empezaremos a utilizar la terminología de la programación orientada a objetos que se presentó en la sección 1.15.

Al igual que en el capítulo 2, hay unos cuantos casos en los que todavía no proporcionamos los detalles necesarios para crear aplicaciones y applets complejos en Java. Es importante que usted aumente primero su conocimiento acerca de los conceptos fundamentales de la programación. En los capítulos 4 y 5 presentamos un análisis detallado del *desarrollo de programas* y del *control de programas* en Java. A medida que avancemos por este libro, iremos presentando muchas aplicaciones y applets substanciales.

3.2 Applets de muestra incluidos en el Kit de desarrollo de software para Java 2

Comencemos por considerar varios applets de muestra que se incluyen con el Kit de desarrollo de software para Java 2 (J2SDK), versión 1.4.1. Estos applets demuestran una pequeña porción de las poderosas herramientas de Java. Cada uno de los programas de muestra del J2SDK viene también con su *código fuente* en java (los archivos `.java` que contienen los programas para estos applets de Java). Este código fuente le será útil, a medida que aumente su conocimiento sobre Java: podrá leer el código fuente que se proporciona para aprender nuevas y excitantes características de Java. Recuerde, al principio todos los programadores aprenden nuevos conceptos de programación copiando el uso de esos conceptos en los programas existentes. El J2SDK incluye

1. La versión en inglés de Netscape 7 está disponible actualmente en `channels.netscape.com/ns/download.jsp`. La versión en español está disponible en `www.aola.com/netscape/download/`.

muchos programas de ejemplo, y hay una enorme cantidad de recursos de Java en Internet y World Wide Web, los cuales incluyen código fuente en Java.

Los programas de demostración que se proporcionan con el J2SDK se encuentran en su directorio de instalación del J2SDK, en un subdirectorio llamado `demo`. En la versión 1.4 del Kit de desarrollo de software para Java 2, la ubicación predeterminada del directorio `demo` en Windows es

```
c:\j2sdk1.4.1\demo
```

En UNIX/Linux/Mac OS X, es el directorio en el que usted haya instalado el J2SDK, seguido de `j2sdk1.4.1/demo`. Por ejemplo,

```
/usr/local/j2sdk1.4.1/demo
```

En las demás plataformas hay una estructura similar de directorios (o carpetas). Este capítulo supone que el J2SDK está instalado en `c:\j2sdk1.4.1` en Windows, y en su directorio personal `~/j2sdk1.4.1` en UNIX/Linux/Mac OS X.[2]

Si está utilizando una herramienta de desarrollo en Java que no incluye los demos de Java de Sun, puede descargar el J2SDK (con los demos) del sitio Web Java de Sun Microsystems:

```
java.sun.com/j2se/1.4.1/download.html
```

Applet TicTacToe

El applet de demostración `TicTacToe` (también conocido como gato) le permite jugar a usted contra la computadora. Para ejecutar este applet, abra una ventana de comandos (símbolo de MS-DOS en Windows 95/98/ME, símbolo del sistema en Windows NT/2000/XP o una ventana de terminal/shell en UNIX/Linux/Mac OS X) y vaya al directorio `demo` del J2SDK. Cada uno de los sistemas operativos que se mencionan aquí utiliza el comando `cd` para *cambiar directorios*. Por ejemplo:

```
cd c:\j2sdk1.4.1\demo
```

cambia al directorio `demo` en Windows, y

```
cd ~/j2sdk1.4.1/demo
```

cambia al directorio `demo` en UNIX/Linux/Mac OS X.

El directorio `demo` contiene varios subdirectorios. Puede ver estos directorios escribiendo el comando `dir` en la ventana de comandos en Windows, o el comando `ls` en UNIX/Linux/Mac OS X. Nos enfocaremos en los directorios `applets` y `jfc`. El directorio `applets` contiene muchos applets de demostración. El directorio `jfc` (Java Foundation Classes, Clases Fundamentales de Java) contiene muchos ejemplos de gráficos en Java y características de GUI (algunos de estos ejemplos son también applets). Para cambiar al directorio `applets` debe escribir el comando

```
cd applets
```

ya sea en Windows o UNIX/Linux/Mac OS X.

Enliste el contenido del directorio `applets` para ver los nombres de directorio de los applets de demostración. La figura 3.1 proporciona una breve descripción de cada ejemplo.

Ejemplo	Descripción
`Animator`	Realiza una de cuatro animaciones separadas.
`ArcTest`	Demuestra cómo dibujar arcos. Puede interactuar con el applet para cambiar los atributos del arco que se muestra.
`BarChart`	Dibuja un gráfico de barras simple.

Figura 3.1 Los ejemplos del directorio `applets`. (Parte 1 de 2.)

2. Tal vez necesite actualizar estas ubicaciones para usar el directorio de instalación y la unidad de disco que usted seleccionó, o una versión distinta del J2SDK.

http://librosysolucionarios.net

Ejemplo	Descripción
`Blink`	Muestra texto parpadeante en distintos colores.
`CardTest`	Muestra varios componentes de GUI y una variedad de maneras en las que pueden ordenarse en la pantalla. (El orden de los componentes de GUI se conoce también como la *distribución* de los componentes de GUI.)
`Clock`	Dibuja un reloj con "manecillas" giratorias, la fecha y hora actuales. El reloj se actualiza una vez por segundo.
`DitherTest`	Muestra cómo dibujar con una técnica de gráficos conocida como difuminado, la cual permite una transformación gradual de un color a otro.
`DrawTest`	Permite al usuario arrastrar el ratón para dibujar líneas y puntos en el applet, en distintos colores.
`Fractal`	Dibuja un fractal. Los fractales generalmente requieren de cálculos complejos para determinar la manera en que van a mostrarse.
`GraphicsTest`	Dibuja una variedad de figuras para ilustrar las capacidades de los gráficos.
`GraphLayout`	Dibuja un gráfico que consiste de muchos nodos (representados como rectángulos) conectados por líneas. Arrastre uno de los nodos y verá cómo los demás nodos en el gráfico se ajustan en la pantalla, para demostrar interacciones gráficas complejas.
`ImageMap`	Muestra una imagen con *puntos activos*. Al colocar el puntero del ratón sobre ciertas áreas de la imagen, se resalta el área y se muestra un mensaje en la esquina inferior izquierda de la ventana del `appletviewer`. Colóquese sobre la boca para escuchar cómo la imagen dice "hola".
`JumpingBox`	Desplaza un rectángulo en forma aleatoria, alrededor de la pantalla. ¡Trate de atraparlo haciendo clic sobre él con el ratón!
`MoleculeViewer`	Presenta una vista tridimensional de varias moléculas químicas distintas. Arrastre el ratón y verá la molécula desde varios ángulos.
`NervousText`	Arrastra texto que salta por la pantalla.
`SimpleGraph`	Dibuja una curva compleja.
`SortDemo`	Compara tres técnicas de ordenamiento. El ordenamiento (que se describe en el capítulo 7) sirve para organizar la información; es como alfabetizar las palabras. Cuando usted ejecuta el applet, aparecen tres ventanas del `appletviewer`. Haga clic en cada una de ellas para empezar con el ordenamiento. Observe que cada una de las tres técnicas de ordenamiento operan a distintas velocidades.
`SpreadSheet`	Muestra una hoja de cálculo simple, con filas y columnas.
`SymbolTest`	Dibuja caracteres del conjunto de caracteres de Java.
`TicTacToe`	Permite al usuario jugar contra la computadora.
`WireFrame`	Dibuja una figura tridimensional como malla alámbrica. Arrastre el ratón para ver la figura desde varios ángulos.

Figura 3.1 Los ejemplos del directorio `applets`. (Parte 2 de 2.)

Cambie ahora al subdirectorio `TicTacToe`. En ese directorio encontrará el documento HTML `example1.html` que se utiliza para ejecutar el applet. En la ventana de comandos, escriba el comando

```
appletviewer example1.html
```

y oprima la tecla *Entrar*. Esto hace que se ejecute el contenedor de applets `appletviewer`, el cual carga el documento HTML `example1.html` que se especifica como su *argumento de línea de comandos*, determina en base al archivo qué applet debe cargar[3] y comienza a ejecutarlo. La figura 3.2 muestra varias capturas de pantalla del juego de TicTacToe con este applet.

3. Hablaremos sobre los detalles de esto en la sección 3.3

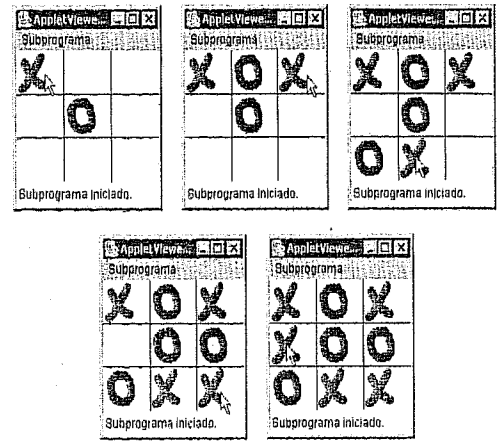


Figura 3.2 Ejemplo de la ejecución del applet `TicTacToe`.

Tip para prevenir errores 3.1

Si el comando `appletviewer` *no funciona o recibe un mensaje indicando que el comando* `appletviewer` *no puede encontrarse, tal vez la variable de entorno* `PATH` *no esté correctamente definida en su computadora. Revise las indicaciones de instalación para el Kit de desarrollo de software para Java 2, para asegurarse de que la variable de entorno* `PATH` *esté definida correctamente en su sistema. Tal vez necesite reiniciar su equipo después de modificar la variable* `PATH`.

Usted es el jugador **X**. Para interactuar con el applet, coloque el ratón sobre el cuadro en el que desea colocar una **X** y haga clic con el botón del ratón. El applet reproduce un sonido (suponiendo que su computadora tenga soporte para la reproducción de audio) y coloca una X en el cuadro, si éste está libre. Si el cuadro está ocupado, es un movimiento inválido y el applet reproduce un sonido distinto, indicando que usted no puede hacer el movimiento especificado. Después de que haga un movimiento válido, el applet responderá con su propio movimiento.

Para jugar de nuevo, haga clic en el *menú* **Subprograma (Applet)** del `appletviewer` y seleccione el *elemento de menú* **Volver a cargar (Reload)** (figura 3.3). Para terminar el `appletviewer`, haga clic en el menú **Subprograma** y seleccione el *elemento de menú* **Salir (Quit)**.

Applet DrawTest

El applet de demostración `DrawTest` le permite dibujar líneas y puntos en distintos colores. En la ventana de comandos, cambie al directorio `applets` y después al subdirectorio `DrawTest`. Puede desplazarse hacia arriba del árbol de directorios para llegar a `demo`, mediante el comando `"cd..."`, tanto en Windows como en UNIX/Linux/Mac OS X. En el directorio `DrawTest` se encuentra el documento `example1.html` que se utiliza para ejecutar el applet. En la ventana de comandos, escriba el comando

```
appletviewer example1.html
```

y oprima la tecla *Entrar*. Esto hace que se ejecute el `appletviewer`, que a su vez carga el documento HTML especificado como su argumento de línea de comandos (`example1.html` de nuevo), determina en base a este archivo qué applet cargar y comienza a ejecutarlo. La figura 3.4 muestra una captura de pantalla de este applet, después de dibujar algunas líneas y puntos.

La figura predeterminada para dibujar es una línea y el color predeterminado es el negro, por lo que puede dibujar líneas negras inmediatamente, con sólo *arrastrar el ratón* a lo largo del applet. Por lo tanto, mantenga presionado el botón del ratón mientras lo arrastra para hacer el dibujo. Observe que la línea sigue al puntero del ratón alrededor del applet. La línea no será permanente sino hasta que usted suelte el botón del ratón. Entonces puede empezar una nueva línea con sólo repetir el proceso.

http://librosysolucionarios.net

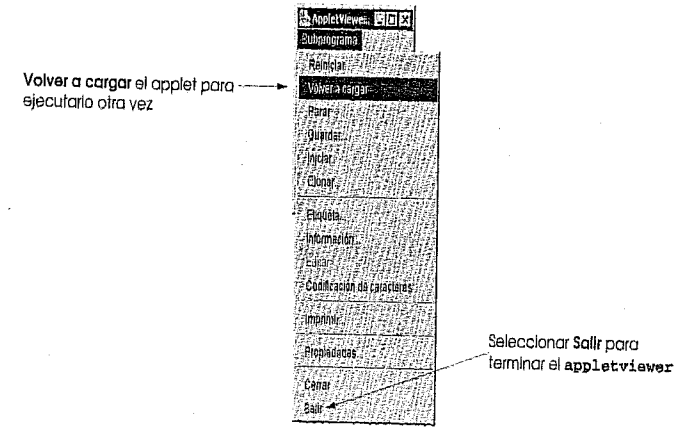


Figura 3.3 Menú **Subprograma** del `appletviewer`.

Seleccione un color haciendo clic en el círculo que está dentro de uno de los rectángulos de colores, en la parte inferior del applet. Puede seleccionar de entre rojo, verde, azul, rosa, naranja y negro. Los componentes de GUI que se utilizan para presentar estas opciones se conocen comúnmente como *botones de opción*. Por ejemplo, en el radio de un automóvil, sólo puede seleccionarse una estación a la vez. Lo mismo ocurre en este caso, sólo puede seleccionar un color a la vez.

Cambie la forma para dibujar de líneas (**Lines**) a puntos (**Points**), haciendo clic en la flecha hacia abajo que se encuentra a la derecha de la palabra **Lines**, en la parte inferior del applet. A continuación se despliega una lista del componente de GUI, la cual contiene las dos opciones: **Lines** y **Points**. Para seleccionar **Points**, haga

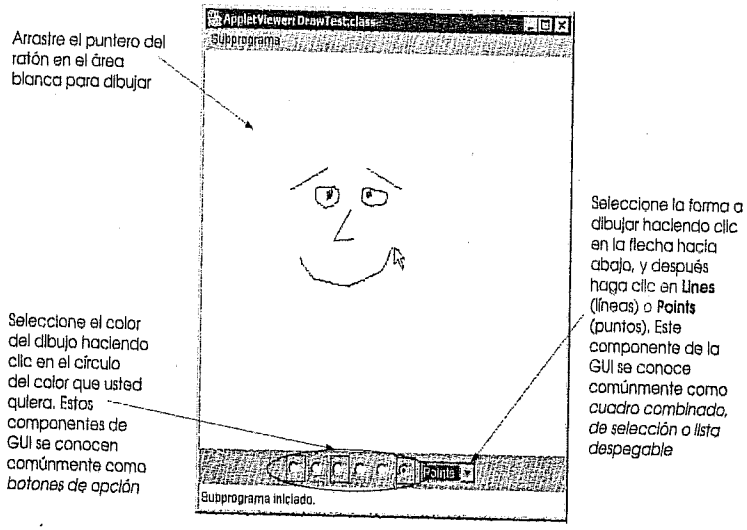


Figura 3.4 Ejemplo de la ejecución del applet `DrawTest`.

http://librosysolucionarios.net

clic en esta palabra en la lista. El componente de GUI cierra la lista y el tipo de forma actual es ahora **Points**. Este componente de GUI se conoce como *cuadro combinado, de selección* o *lista desplegable*.

Para empezar un nuevo dibujo, seleccione la opción **Volver a cargar (Reload)** en el menú **Subprograma** del `appletviewer`. Para terminar el applet, seleccione **Salir (Quit)** en el menú **Subprograma** del `appletviewer`.

Applet Java2D

Este applet muestra el *API Java2D* (las herramientas de Java para dibujar y manipular gráficos bidimensionales complejos). Cambie al directorio `jfc` que se encuentra en el directorio `demo` del J2SDK, y después cambie al directorio `Java2D`. En ese directorio se encuentra el documento HTML `Java2Demo.html`, el cual se utiliza para cargar y ejecutar el applet. En la ventana de comandos, escriba el comando

```
appletviewer Java2Demo.html
```

para cargar el documento HTML `Java2Demo.html` y empezar la ejecución del applet. Esta demostración tarda algo de tiempo en cargar, ya que es bastante grande. La figura 3.5 muestra una captura de pantalla de una de las muchas demostraciones de este applet, en relación con las herramientas para gráficos bidimensionales de Java.

En la parte superior del applet hay fichas que parecen carpetas en un archivero. Esta demostración cuenta con 12 fichas, cada una de las cuales tiene varias características. Para cambiar a una parte distinta de la demostración, simplemente haga clic en una de las fichas. También puede probar cambiando las opciones en la esquina superior derecha del applet. Algunas de estas opciones afectan la velocidad con la que el applet dibuja los gráficos. Por ejemplo, haga clic en el pequeño cuadro con una marca de verificación (un componente de GUI conocido como *casilla de verificación*) que está a la izquierda de la palabra **Anti-Aliasing** para activar y desactivar el suavizado (una técnica de gráficos para producir gráficos en pantalla más suaves, en los que los bordes del gráfico están desenfocados). Cuando esta característica se desactiva (es decir, su *casilla de verificación* se deselecciona), la velocidad de animación aumenta para las figuras animadas que están en la parte inferior de la demostración, como se muestra en la figura 3.5. Esto ocurre debido a que una figura animada que se muestra con suavizado tarda más en dibujarse que una figura animada sin suavizado.

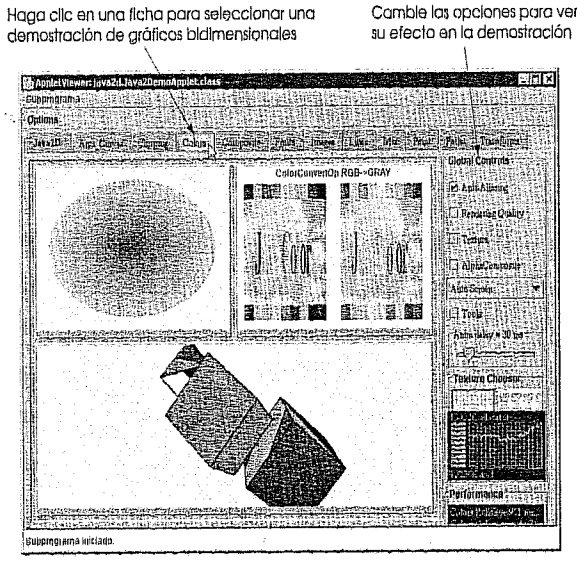


Figura 3.5 Ejemplo de la ejecución del applet `Java2D`.

3.3 Applet simple en Java: Cómo dibujar una cadena

Ahora crearemos nuestros propios applets. Recuerde, sólo estamos empezando; tenemos muchos temas que aprender antes de poder escribir applets similares a los que se muestran en la sección 3.2. Sin embargo, cubriremos muchas de las mismas técnicas en este libro.

Comenzaremos con un applet simple que dibuja **"¡Bienvenido a la programación en Java!"** El applet y su salida en pantalla se muestran en la figura 3.6. La pantalla muestra a este applet ejecutándose en tres contenedores de applets: el **appletviewer**, el navegador Web Netscape y el navegador Web Microsoft Internet Explorer. Al final de esta sección explicaremos cómo ejecutar el applet en un navegador Web.

```
1  // Fig. 3.6: SubprogramaBienvenido.java
2  // Nuestro primer applet en Java.
3
4  // Paquetes de Java
5  import java.awt.Graphics;    // importar la clase Graphics
6  import javax.swing.JApplet;  // importar la clase JApplet
7
8  public class SubprogramaBienvenido extends JApplet {
9
10    // dibujar texto en el fondo del applet
11    public void paint( Graphics g )
12    {
13       // llamar a la versión del método paint de la superclase
14       super.paint( g );
15
16       // dibujar un String en la coordenada x 25 y la coordenada y 25
17       g.drawString( "¡Bienvenido a la programación en Java!", 25, 25 );
18    } // fin del método paint
19
20 } // fin de la clase SubprogramaBienvenido
21
```

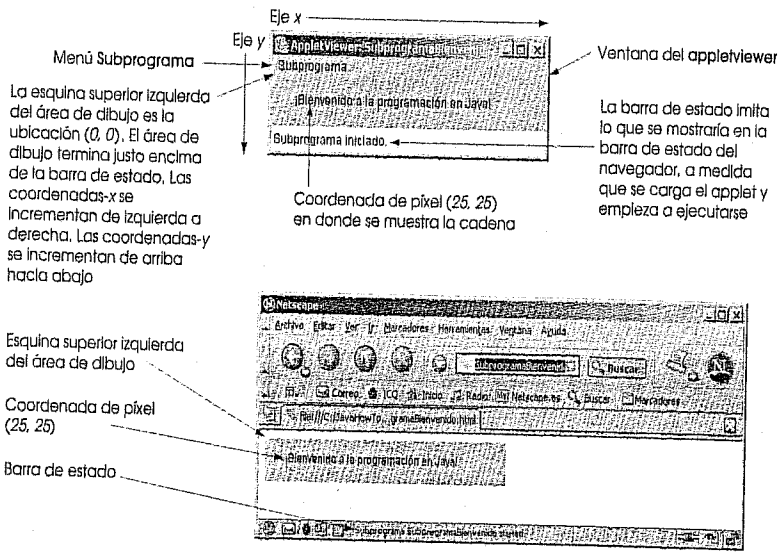


Figura 3.6 Applet que dibuja una cadena. (Parte 1 de 2.)

http://librosysolucionarios.net

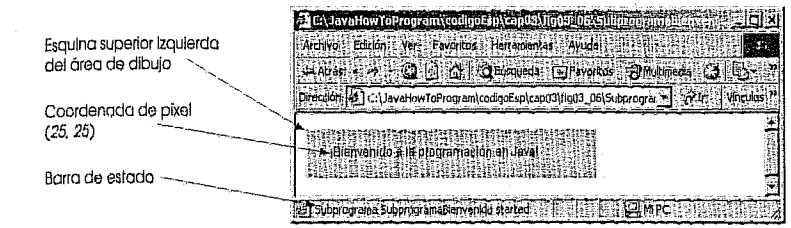


Figura 3.6 Applet que dibuja una cadena. (Parte 2 de 2.)

Este programa muestra varias características importantes de Java. Consideraremos cada una de las líneas del programa detalladamente. La línea 17 hace el "verdadero trabajo" del programa; dibujar la cadena **¡Bienvenido a la programación en Java!** en la pantalla. Pero consideremos cada línea del programa en orden. Las líneas 1 y 2

```
// Fig. 3.6: SubprogramaBienvenido.java
// Nuestro primer applet en Java.
```

son comentarios. La línea 1 indica el número de figura y el nombre de archivo para el código fuente del applet. La línea 2 simplemente describe el propósito del programa.

Recuerde del capítulo 2 que Java contiene muchas entidades predefinidas, llamadas clases, que se agrupan en paquetes (colecciones de clases con nombres) en el API de Java. La línea 5

```
import java.awt.Graphics;    // importar la clase Graphics
```

es una declaración **import** que indica que el applet utiliza la clase ***Graphics*** del paquete **java.awt.** Esta clase permite a un applet de Java dibujar gráficos, como líneas, rectángulos, óvalos y cadenas de caracteres.[4]

La línea 6

```
import javax.swing.JApplet;  // importar la clase JApplet
```

es una declaración **import**, la cual indica que el applet utiliza la clase ***JApplet*** del paquete **javax.swing**. Al crear un applet en Java 2, debe importarse la clase **JApplet**.[5]

Al igual que con las aplicaciones, todos los applets de Java contienen cuando menos una declaración de clase pública. Una característica clave de las declaraciones de clases es que los programadores raras veces crean declaraciones de clases "a partir de cero". De hecho, cuando se crea una declaración de clase, normalmente se utilizan piezas de una declaración de clase existente. Java utiliza la *herencia* (este término se presentó en la sección 1.15, y se describe con detalle en el capítulo 9, Programación orientada a objetos) para crear nuevas clases a partir de las declaraciones de clases existentes. La línea 8

```
public class SubprogramaBienvenido extends JApplet {
```

comienza la declaración **class** para la clase **SubprogramaBienvenido**. El cuerpo de la clase está delimitado por la llave izquierda, **{**, en la línea 8 y su correspondiente llave derecha, **}**, en la línea 21. La palabra clave **class** introduce la declaración de la clase. **SubprogramaBienvenido** es el nombre de la clase. La palabra clave ***extends*** indica que la clase **SubprogramaBienvenido** *hereda* los miembros existentes (datos y métodos) de otra clase. La clase de la que hereda **SubprogramaBienvenido**, **JApplet**, aparece a la derecha de la palabra clave **extends**. En esta relación de herencia, **JApplet** se conoce como la *superclase* y **SubprogramaBienvenido** se conoce como la *subclase*.[6] El uso de la herencia aquí produce una clase **SubprogramaBienvenido** que tiene los atributos (datos) y comportamientos (métodos) de la clase **JApplet**,

4. El capítulo 12 muestra cómo dibujar en una aplicación.
5. Hay una clase más antigua llamada **Applet** (del paquete **java.applet**) que no es compatible con los componentes de GUI de Java del paquete **javax.swing**. Todas las clases de los applets en este libro extienden la clase **JApplet**.
6. En algunos lenguajes, las superclases y subclases se conocen como *clases base* y *clases derivadas*.

así como las nuevas características que agregaremos en nuestra declaración de la clase `SubprogramaBienvenido` (específicamente, la habilidad de dibujar `¡Bienvenido a la programación en Java!` en el applet).

Nosotros extendemos la clase `JApplet` porque alguien más definió anteriormente "lo que significa ser un applet." Los contenedores de applets esperan que todos los applets de Java tengan ciertos comportamientos (métodos.) La clase `JApplet` ya proporciona todos esos comportamientos. De hecho, un contenedor de applets espera que cada applet que ejecute tenga más de 200 métodos distintos. Hasta este punto, en nuestros programas definimos un método en cada uno. Si tuviéramos que definir más de 200 métodos sólo para mostrar `¡Bienvenido a la programación en Java!` nunca crearíamos un applet, ¡ya que esto tomaría demasiado tiempo! El uso de `extends` para heredar de la clase `JApplet` nos permite crear nuevos applets rápidamente, al definir solamente lo que es nuevo y distinto acerca de nuestros applets.

El mecanismo de herencia es fácil de utilizar; el programador no necesita conocer todos los detalles acerca de la clase `JApplet` ni de cualquier otra superclase que vaya a heredar a una nueva clase. El programador sólo necesita saber que la clase `JApplet` define las capacidades requeridas para crear el applet mínimo. Sin embargo y para utilizar una clase lo mejor posible, los programadores deben estudiar todas las capacidades de la superclase.



Buena práctica de programación 3.1

Investigue cuidadosamente las capacidades de una clase en la documentación para el API de Java (`java.sun.com/j2se/1.4/docs/api/index.html`) antes de extender la clase para crear una subclase. Esto le ayudará a asegurarse de no redefinir inadvertidamente una capacidad que la superclase ya proporcione.

Las clases se utilizan como "plantillas" o "planos de construcción" para *instanciar* (o crear) *objetos*, de manera que puedan usarse en un programa. Un objeto (o *instancia* o *instancia de clase*)[7] reside en la memoria de la computadora y contiene información utilizada por el programa. El término *objeto* normalmente implica que los atributos (datos) y comportamientos (métodos) están asociados con el objeto. Los métodos del objeto utilizan sus atributos para proporcionar servicios útiles al *cliente del objeto* (es decir, el código en un programa que llama a los métodos).

Cuando un contenedor de applets carga la clase `SubprogramaBienvenido`, el contenedor crea un objeto de tipo `SubprogramaBienvenido` que implementa los atributos y comportamientos del applet. El contenedor de applets es el cliente del objeto `SubprogramaBienvenido`; es decir, el contenedor de applets llamará a los métodos de `SubprogramaBienvenido`. Los contenedores de applets pueden crear sólo objetos de clases que sean `public` y que extiendan a `JApplet`.[8] Por lo tanto, los contenedores de applets requieren que las declaraciones de clases de los applets empiecen con la palabra clave `public`[9] (línea 8). De no ser así, el contenedor de applets no podrá cargar ni ejecutar el applet.

Recuerde del capítulo 2 que, cuando se guarda una clase `public` en un archivo, el nombre del mismo debe ser el nombre de la clase seguido por la extensión `.java`. Para nuestro applet, el nombre de archivo debe ser `SubprogramaBienvenido.java`. Para reforzar conceptos, repetiremos dos errores comunes de programación del capítulo 2.



Error común de programación 3.1

Tanto en términos de ortografía como en ahorro de tiempo, es un error que una clase `public` tenga un nombre de archivo que no sea idéntico al nombre de la clase (más la extensión `.java`). Por lo tanto, es también un error el que un archivo contenga dos o más clases `public`.



Error común de programación 3.2

Si un archivo contiene la declaración de una clase, es un error que no finalice con la extensión `.java`. Si se omite esa extensión, el compilador de Java no podrá compilar la declaración de la clase.

7. Los términos instancia, instancia de clase y objeto se utilizan a menudo para definir lo mismo.
8. En realidad, los contenedores de applets pueden crear también objetos de clases que extiendan la clase `Applet`, proveniente de versiones más antiguas de Java.
9. El capítulo 8, Programación orientada a objetos, habla detalladamente sobre la palabra clave `public` y las palabras clave relacionadas (como `private` y `protected`). Por ahora, comience todas sus declaraciones de clase con `public`.

http://librosysolucionarios.net



Tip para prevenir errores 3.2

El mensaje de error del compilador "`Public class NombreClase must be defined in a file called NombreClase.java`" indica que el nombre de archivo no concuerda exactamente con el nombre de la clase `public` en el archivo, o que usted escribió el nombre de clase incorrectamente, al compilar la clase.

La línea 11

```
public void paint( Graphics g )
```

comienza la declaración del *método* `paint` del applet, que es uno de tres métodos del applet (algunas veces llamados comportamientos) que el contenedor de applets llama cuando éste empieza a ejecutar el applet. En orden, estos métodos son: `init` (lo describiremos más adelante en este capítulo), `start` (descrito en el capítulo 6) y `paint`.[10] La clase de su applet recibe una versión "gratuita" de cada uno de estos métodos de la clase `JApplet`, al especificar `extends JApplet` en la primera línea de la declaración de clase. Si usted no declara estos métodos en su applet, el contenedor de applets llama a las versiones heredadas. Los métodos `init` y `start` de la superclase tienen cuerpos vacíos (es decir, sus cuerpos no contienen instrucciones, por lo que no realizan una tarea), y el método `paint` de la superclase no dibuja nada en el applet.

Para que nuestro applet pueda dibujar, la clase `SubprogramaBienvenido` *sobrescribe (redefine)* la versión de `paint` de la superclase, colocando instrucciones en el cuerpo de `paint` para dibujar un mensaje en la pantalla. Cuando el contenedor de applets indica al applet que "se dibuje a sí mismo" en la pantalla mediante una llamada al método `paint`, aparece nuestro mensaje `¡Bienvenido a la programación en Java!` en vez de una pantalla en blanco.

Las líneas 11 a la 19 declaran el método `paint`, que dibuja gráficos (como líneas, óvalos y cadenas de caracteres) en un applet. La palabra clave `void` (línea 11) indica que este método no devuelve resultados al completar su tarea. El conjunto de paréntesis después de `paint` define la *lista de parámetros* del método, la cual especifica los datos que éste requiere para realizar su tarea. Por lo general, los programadores pasan datos a un método mediante una llamada a ese método (lo que también se conoce como *invocar a un método* o *enviar un mensaje*). En el capítulo 2 pasamos datos (como el mensaje a mostrar) al método `showMessageDialog` de `JOptionPane`, y pasamos datos a `System.out.println` para mostrarlos en la ventana de comandos. Al escribir applets, los programadores no llaman al método `paint` en forma explícita. En vez de eso, el contenedor de applets llama a `paint` para decir al applet que dibuje, y el contenedor de applets pasa un argumento a `paint`: un objeto `Graphics` (conocido como `g`), que `paint` requiere para realizar su tarea. Es responsabilidad del contenedor de applets crear el objeto `Graphics` al que se refiere `g`. El método `paint` utiliza la referencia (`g`) al objeto `Graphics` para dibujar gráficos en el applet. Es necesario que utilice la palabra clave `public` al principio de la línea 11 para que el contenedor de applets pueda llamar al método `paint` de la clase que usted haya creado. Por ahora, todas las declaraciones de métodos (al igual que las declaraciones de clases) deben comenzar con la palabra clave `public`.[11]

El cuerpo del método `paint` está delimitado por la llave izquierda, `{`, en la línea 12. Su correspondiente llave derecha, `}`, en la línea 19, termina el cuerpo de `paint`.

La línea 14

```
super.paint( g );
```

llama a la versión del método `paint` de la superclase `JApplet`.[12] Por ahora, ésta debe ser la primera instrucción en todas las declaraciones del método `paint`.

La línea 17

```
g.drawString( "¡Bienvenido a la programación en Java!", 25, 25 );
```

10. El capítulo 6, Métodos, habla sobre varios métodos más que son llamados por un contenedor de applets durante la ejecución de un applet.
11. Presentaremos otras alternativas en el capítulo 8.
12. Por razones que se aclararán más adelante en el texto, esta instrucción debe ser la primera en el método `paint` de cualquier applet. Aunque los primeros ejemplos de applets funcionan sin esta instrucción, omitirla provoca ligeros errores en applets más elaborados en los que se combinan los dibujos y los componentes de GUI. Si incluye esta instrucción ahora, adquirirá el hábito de usarla, ahorrando tiempo y esfuerzo a medida que construya applets más robustos posteriormente. Explicaremos la sintaxis de `"super"` más adelante, en el capítulo 9.

indica a la computadora que realice una acción (o tarea): dibujar los caracteres de la cadena `¡Bienvenido a la programación en Java!` en el applet. Esta instrucción utiliza el método `drawString` de la clase `Graphics`, el cual proporciona capacidades de dibujo como dibujar cadenas de caracteres, líneas, rectángulos y óvalos. La instrucción llama a `drawString` utilizando la referencia `g` de la lista de parámetros de `paint`, seguida por un *separador de punto* (`.`), al cual le sigue el nombre del método, `drawString`. Este nombre va seguido por un conjunto de paréntesis que contienen los argumentos que `drawString` necesita para realizar su tarea. La `"g."` al principio de la instrucción indica que `paint` debe utilizar el objeto `Graphics` que el contenedor de applets pasa a `paint`.

El primer argumento para `drawString` es la cadena que va a dibujar. Los últimos dos argumentos en la lista (`25` y `25`) son las *coordenadas x-y* (o la *posición*) en donde debe aparecer la esquina inferior izquierda de la cadena. Los métodos para dibujar de la clase `Graphics` requieren coordenadas que les especifiquen en dónde dibujar.[13] La figura 3.7 muestra el sistema de coordenadas de Java. Las coordenadas se miden a partir de la esquina superior izquierda del applet, en *píxeles*. Un píxel ("elemento de imagen") es una unidad de resolución para la pantalla de su computadora. Cada píxel tiene un *par de coordenadas* que describe la posición de ese píxel en la pantalla. En un par de coordenadas, la primera coordenada es la *coordenada x* (el número de píxeles contando a partir del lado izquierdo del applet) y la segunda coordenada es la *coordenada y* (el número de píxeles contando a partir de la parte superior del applet). En el `appletviewer`, la coordenada *(0, 0)* del applet aparece justo debajo del menú **Subprograma** (del `Appletviewer`). En un navegador Web, la coordenada *(0, 0)* del applet aparece en la esquina superior izquierda del área rectangular en la que se ejecuta el applet (vea la figura 3.6).

En una pantalla de computadora, un píxel aparece como un punto coloreado en la pantalla. Muchas computadoras personales tienen 800 píxeles de anchura en la pantalla, y 600 píxeles de altura, para un total de 800 por 600, o 480,000 píxeles que pueden mostrarse en la pantalla. La mayoría de las computadoras son capaces de producir mayores resoluciones (es decir, tienen más píxeles de anchura y de altura en la pantalla). El tamaño de un applet en la pantalla depende del tamaño y la resolución de la misma. En pantallas del mismo tamaño, el applet será más pequeño en la pantalla a una mayor resolución debido a que habrá un mayor número de píxeles en la pantalla.

Al ejecutarse la línea 17, se dibuja la cadena `¡Bienvenido a la programación en Java!` en el applet, en las coordenadas `25` y `25`. Observe que las comillas que encierran a la cadena no aparecen en la pantalla.

Pasando a otra cosa, ¿para qué sirven las copias de los métodos `init`, `start` y `paint` si no realizan una tarea? La secuencia de inicio predefinida de las llamadas a métodos realizadas por el contenedor de applets, para cada applet, siempre es `init`, `start` y `paint`; esto garantiza que estos métodos se llamen al momento en que empiece la ejecución de cualquier applet. No todos los applets necesitan los tres métodos. Sin embargo, el contenedor de applets no sabe eso. Por lo tanto, espera que cada uno de estos métodos esté definido, para poder proporcionar una secuencia de inicio consistente para cada applet. Esto es similar a las aplicaciones que siempre inician su ejecución con `main`. Al heredar las versiones "predeterminadas" de estos métodos, se garantiza que el contenedor de applets pueda tratar a cada applet de la misma forma, llamando a `init`, `start` y `paint` al comenzar a ejecutar un applet. Además, el programador puede concentrarse en definir sólo los métodos requeridos para un applet específico.

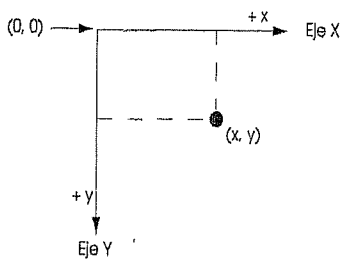


Figura 3.7 El sistema de coordenadas de Java. Las unidades se miden en píxeles.

13. Demostraremos cómo dibujar en las aplicaciones, en el capítulo 12.

Compilación y ejecución de applet `SubprogramaBienvenido`[14]

Al igual que con las clases de las aplicaciones, debe compilar la clase de un applet para poder ejecutarla. Después de crear la clase `SubprogramaBienvenido` y guardarla en `SubprogramaBienvenido.java`, abra una ventana de comandos, cambie al directorio en el que guardó la declaración de clase del applet y escriba el comando

```
javac BienvenidoSubprograma.java
```

para compilar la clase de `SubprogramaBienvenido`. Si no hay errores de sintaxis, los códigos de bytes resultantes se guardan en el archivo `SubprogramaBienvenido.class` (el archivo de clase para `SubprogramaBienvenido`).

Recuerde que los applets se incrustan en páginas Web para ejecutarse en un contenedor de applets (`appletviewer` o un navegador). Antes de que pueda ejecutar el applet, debe crear un documento *HTML* para cargar el applet en el contenedor de applets. Por lo general, un documento HTML termina con la extensión de archivo "`.html`" o "`.htm`". Los navegadores muestran el contenido de los documentos que contienen texto (los que también se conocen como *archivos de texto*). Para ejecutar un applet de Java, un documento HTML debe indicar cuál applet debe cargar y ejecutar el contenedor de applets. La figura 3.8 muestra un documento HTML simple (`SubprogramaBienvenido.html`), el cual carga el applet definido en la figura 3.6, en un contenedor de applets.

Muchos elementos del HTML se delimitan por pares de *etiquetas (tags)*. Por ejemplo, las líneas 1 y 4 de la figura 3.8 indican el inicio y el final, respectivamente, del documento HTML. Todas las etiquetas de HTML comienzan con un *signo menor que*, `<`, y terminan con un *signo mayor que*, `>`. Las líneas 2 y 3 especifican un *elemento* `applet`, el cual indica al contenedor de applets que debe cargar cierto applet, y define el tamaño del área de despliegue del applet (su *anchura* y *altura* en píxeles) en el contenedor de applets (es decir, el `appletviewer` o el navegador). Por lo general, el applet y su documento HTML correspondiente se guardan en el mismo directorio en el disco. Comúnmente, un navegador carga un documento HTML desde una computadora (que no es la de usted) conectada a Internet. Sin embargo, los documentos HTML también pueden residir en su computadora (como lo demostramos en la sección 3.2). Cuando un contenedor de applets se encuentra con un documento HTML que especifica un applet a ejecutar, el contenedor de applets carga automáticamente el archivo (o los archivos) `.class` del applet, desde el mismo directorio en la computadora en la que reside el documento HTML.

El elemento `applet` tiene varios *atributos*. El primer atributo en la línea 2, `code = "SubprogramaBienvenido.class"`, indica que el archivo `SubprogramaBienvenido.class` contiene la clase del applet compilada. Recuerde que, al compilar sus programas en Java, cada clase se compila en un archivo separado que tiene el mismo nombre que la clase y termina con la extensión `.class`. El segundo y tercer atributos en la línea 2 indican la `anchura (width)` y la `altura (height)` del applet, en píxeles. La esquina superior izquierda del área de despliegue del applet siempre tiene la coordenada *x* 0 y la coordenada *y* 0. La anchura de este applet es de 300 píxeles y su altura es de 45 píxeles. Tal vez usted quiera (o necesite) utilizar valores más grandes en `width` y `height`, para definir un área mayor para sus applets. La marca `</applet>` (línea 3) termina el elemento `applet` que empezó en la línea 2. La marca `</html>` (línea 4) especifica el final del documento HTML.

14. En la página `www.deitel.com/books/downloads.html` de nuestro sitio Web contamos con publicaciones de la serie DEITEL™ DIVE INTO™ que le ayudarán a empezar a utilizar varias herramientas populares de desarrollo con Java™, incluyendo el Kit de desarrollo de software para Java™ 2 de Sun™ Microsystems versión 1.4, el software Sun One Studio 4 Community Edition de Sun Microsystems, y Borland® JBuilder™ 7 Personal. Tendremos otras publicaciones de la serie DEITEL™ DIVE INTO™ disponibles, a medida que los maestros las soliciten.

http://librosysolucionarios.net

```
1 <html>
2 <applet code = "SubprogramaBienvenido.class" width = "300" height = "45">
3 </applet>
4 </html>
```

Figura 3.8 `SubprogramaBienvenido.html` carga la clase `SubprogramaBienvenido` de la figura 3.6 en el `appletviewer`.

Observación de apariencia visual 3.1

Para asegurar que un applet pueda verse apropiadamente en la mayoría de las pantallas de computadora, cada applet debe ser menor de 800 píxeles de ancho y 600 píxeles de largo (medidas soportadas por la mayoría de las pantallas de computadora).

Error común de programación 3.3

Olvidar la marca `</applet>` de finalización evita que el applet se cargue apropiadamente en el `appletviewer` o navegador.

Tip para prevenir errores 3.3

Si recibe un mensaje de error `MissingResourceException` al cargar un applet en el `appletviewer` o navegador, verifique cuidadosamente la marca `<applet>` en el documento HTML para asegurarse de que no haya errores de sintaxis, como comas (`,`) entre los atributos.

El `appletviewer` comprende solamente las etiquetas de HTML `<applet>` y `</applet>`, por lo que algunas veces se conoce como el "navegador mínimo". (Ignora todas las demás etiquetas de HTML.) El `appletviewer` es un lugar ideal para probar un applet y asegurarse de que se ejecute correctamente. Una vez que se haya verificado la ejecución del applet, podrá agregar las etiquetas de HTML del applet a un documento HTML que será visto por las personas que naveguen en Internet.

Para ejecutar `SubprogramaBienvenido` en el `appletviewer`, abra una ventana de comandos, cambie al directorio que contiene su applet y su documento HTML, y después escriba

```
appletviewer SubprogramaBienvenido.html
```

El `appletviewer` utiliza un documento HTML para cargar un applet. Esto difiere del intérprete `java` para las aplicaciones, el cual requiere sólo el nombre de la clase de la aplicación. Además, el comando anterior debe ejecutarse desde el directorio en donde se encuentran el documento HTML y el archivo `.class` del applet.

Tip para prevenir errores 3.4

Pruebe sus applets en el contenedor de applets (`appletviewer`) antes de ejecutarlos en un navegador Web. Esto le permitirá ver los mensajes de error que puedan ocurrir. Además, una vez que se ejecuta un applet en un navegador, algunas veces es difícil volver a cargar el applet después de hacer modificaciones a la declaración de clase del applet. Los navegadores a menudo guardan una copia de un applet en memoria, hasta que termina la sesión actual de navegación (es decir, cuando se cierran todas las ventanas del navegador). Por lo tanto, si usted modifica un applet, lo vuelve a compilar y luego lo carga en el navegador, tal vez no vea los cambios debido a que el navegador puede aún estar ejecutando la versión original del applet. Cierre todas las ventanas de su navegador para eliminar de memoria la versión antigua del applet. Abra una nueva ventana del navegador y cargue el applet para ver sus cambios.

Tip para prevenir errores 3.5

Pruebe sus applets en todos los navegadores Web (incluyendo varias versiones del mismo navegador Web) en los que los applets vayan a ejecutarse, para asegurarse de que operen correctamente en cada uno de los navegadores.

Cómo ejecutar un applet en un navegador Web

Las ejecuciones del programa de muestra en la figura 3.6 muestran a `SubprogramaBienvenido` ejecutándose en el `appletviewer` y en los navegadores Web, Netscape y Microsoft Internet Explorer. Para ejecutar un applet en Netscape, realice los siguientes pasos:

1. Seleccione **Open Page...** del menú **File**.

2. En el cuadro de diálogo que se despliega, haga clic en el botón `Choose file...` y localice el directorio que contiene el archivo `.class` y el documento HTML para el applet que usted desea ejecutar.
3. Seleccione el documento HTML.
4. Haga clic en el botón **Abrir** y posteriormente en **Open**.

Para ejecutar un applet en Microsoft Internet Explorer, realice los siguientes pasos:

1. Seleccione **Abrir...** del menú **Archivo**.
2. En el cuadro de diálogo que aparece, haga clic en el botón **Examinar...**.
3. En el siguiente cuadro de diálogo localice el directorio que contenga el archivo `.class` y el documento HTML para el applet que desee ejecutar.
4. Seleccione el documento HTML.
5. Haga clic en el botón **Abrir** y posteriormente en **Aceptar**.

Si su applet se ejecuta en el `appletviewer` pero no en Netscape o Internet Explorer, tal vez no esté instalado el Complemento Java (Java Plug-in). En este caso, visite el sitio Web `java.sun.com/getjava/es/` en español (`java.sun.com/getjava` en inglés). Este sitio proporciona las instrucciones de instalación del Complemento Java para varias plataformas. Si usted utiliza Internet Explorer, simplemente haga clic en el botón **Descárguelo ya. Es gratuito** para instalar el Complemento Java para Internet Explorer.[15]

3.4 Cómo dibujar cadenas y líneas

Consideremos otro applet. Un applet puede dibujar `¡Bienvenido a la programación en Java!` de varias formas. Por ejemplo, puede llamar al método `drawString` dos veces en el método `paint` para mostrar varias líneas de texto, como en la figura 3.9. El documento HTML para cargar este applet en un contenedor de applets se muestra en la figura 3.10.

```
1  // Fig. 3.9: SubprogramaBienvenido2.java
2  // Mostrar varias cadenas en un applet.
3
4  // paquetes de Java
5  import java.awt.Graphics;     // importar la clase Graphics
6  import javax.swing.JApplet;   // importar la clase JApplet
7
8  public class SubprogramaBienvenido2 extends JApplet {
9
10    // dibujar texto en el fondo del applet
11    public void paint( Graphics g )
12    {
13       // llamar a la versión del método paint de la superclase
14       super.paint( g );
15
16       // dibujar dos valores String en diferentes ubicaciones
17       g.drawString( "¡Bienvenido a", 25, 25 );
18       g.drawString( "la programación en Java!", 25, 40 );
```

Figura 3.9 Dibujar cadenas en un applet. (Parte 1 de 2.)

15. Si después de instalar el Complemento Java, los applets todavía no pueden ejecutarse en Internet Explorer, haga clic en el menú **Herramientas** y seleccione **Opciones de Internet...**, después haga clic en la ficha **Opciones avanzadas**, en la ventana que aparecerá. Localice la opción **"Utilice Java 2 v1.4.1 para <applet> (requiere iniciar)"** y asegúrese de que esté activada, después haga clic en **Aceptar**. Cierre todas las ventanas de su navegador antes de intentar cargar otro applet.

http://librosysolucionarios.net

```
19
20     } // fin del método paint
21
22  } // fin de la clase SubprogramaBienvenido2
```

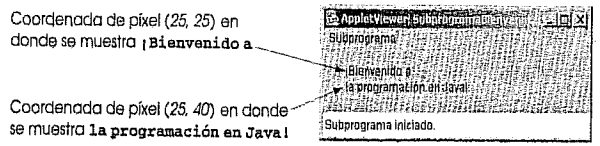


Figura 3.9 Dibujar cadenas en un applet. (Parte 2 de 2.)

```
1  <html>
2  <applet code = "SubprogramaBienvenido2.class" width = "300" height = "60">
3  </applet>
4  </html>
```

Figura 3.10 `SubprogramaBienvenido2.html` carga la clase `SubprogramaBienvenido2` de la figura 3.9, en el `appletviewer`.

Cada llamada a `drawString` puede dibujar en cualquier ubicación de píxel en el applet. La razón por la que las dos líneas de texto están alineadas hacia la izquierda en la figura 3.9 es que ambos utilizan la misma coordenada *x* (`25`). Además, cada llamada al método `drawString` utiliza una coordenada *y* distinta (`25` en la línea 17 y `40` en la línea 18), por lo que las cadenas aparecen en distintas ubicaciones verticales en el applet. Si invertimos las líneas 17 y 18 en el programa, la ventana de salida aún aparecerá como se muestra, ya que el orden de las instrucciones de dibujo no afecta a las coordenadas de píxel especificadas en cada instrucción. Al dibujar gráficos, las líneas de texto no van separadas por caracteres de nueva línea (como era el caso con el método `println` de `System.out` y con el método `showMessageDialog` de `JOptionPane` en el capítulo 2). De hecho, si usted trata de mostrar una cadena que contenga un carácter de nueva línea (`\n`), simplemente verá un pequeño cuadro negro en la posición de la nueva línea en la cadena, lo cual indica que `drawString` no entiende el carácter.

Para hacer el dibujo más interesante, el applet de la figura 3.11 dibuja dos líneas y una cadena. El documento HTML para cargar el applet en un contenedor de applets se muestra en la figura 3.12.

```
1  // Fig. 3.11: LineasBienvenido.java
2  // Mostrar texto y líneas
3
4  // Paquetes de Java
5  import java.awt.Graphics;     // importar la clase Graphics
6  import javax.swing.JApplet;  // importar la clase JApplet
7
8  public class LineasBienvenido extends JApplet {
9
10    // dibujar líneas y una cadena en el fondo del applet
11    public void paint( Graphics g )
12    {
13       // llamar a la versión del método paint de la superclase
14       super.paint( g );
```

Figura 3.11 Dibujar cadenas y líneas. (Parte 1 de 2.)

http://librosysolucionarios.net


```
15
16       // dibujar una línea horizontal, desde (15, 10) hasta (250, 10)
17       g.drawLine( 15, 10, 250, 10 );
18
19       // dibujar una línea horizontal desde (15, 30) hasta (250, 30)
20       g.drawLine( 15, 30, 250, 30 );
21
22       // dibujar String entre líneas, en la ubicación (25, 25)
23       g.drawString( "¡Bienvenido a la programación en Java!", 25, 25 );
24
25    } // fin del método paint
26
27 } // fin de la clase LineasBienvenido
```

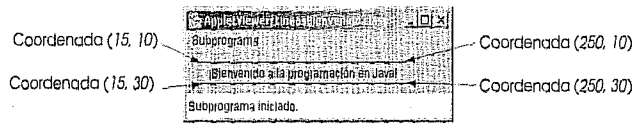


Figura 3.11 Dibujar cadenas y líneas. (Parte 2 de 2.)

```
1  <html>
2  <applet code = "LineasBienvenido.class" width = "300" height = "40">
3  </applet>
4  </html>
```

Figura 3.12 El documento `LineasBienvenido.html`, carga la clase `LineasBienvenido` de la figura 3.11 en el `appletviewer`.

Las líneas 17 y 20 del método `paint`

```
g.drawLine( 15, 10, 210, 10 );
g.drawLine( 15, 30, 210, 30 );
```

usa el *método* `drawLine` de la clase `Graphics` para indicar que el objeto `Graphics`, al que `g` hace referencia, debe dibujar líneas. El método `drawLine` requiere cuatro argumentos que representan los dos puntos finales de la línea en el applet: la coordenada *x* y la coordenada *y* del primer punto final de la línea, y la coordenada *x* y la coordenada *y* del segundo punto final de la línea. Todos los valores de las coordenadas se especifican con respecto a la coordenada *(0, 0)* de la esquina superior izquierda del applet. El método `drawLine` dibuja una línea recta entre los dos puntos finales. En este ejemplo, experimentamos con varios conjuntos de coordenadas para determinar la ubicación y la longitud de las líneas.

3.5 Cómo agregar números de punto flotante

Nuestro siguiente applet (figura 3.13) imita la aplicación para agregar dos números, de la figura 2.9. Sin embargo, este applet solicita que el usuario introduzca dos *números de punto flotante* (es decir, números con un punto decimal, como 7.33, 0.0975 y 1000.12345). Para guardar números de punto flotante en la memoria presentamos el tipo primitivo `double`, el cual representa *números de punto flotante con precisión doble*. También existe el tipo primitivo `float` para guardar *números de punto flotante con precisión simple*. Un `double` requiere más memoria para guardar un valor de punto flotante, pero lo guarda con aproximadamente el doble de precisión de un `float` (15 dígitos significativos para `double`, en comparación con siete dígitos significativos para `float`).

```
1  // Fig. 3.13: SubprogramaSuma.java
2  // Suma dos números de punto flotante.
3
4  // Paquetes de Java
5  import java.awt.Graphics;    // importar la clase Graphics
6  import javax.swing.*;        // importar el paquete javax.swing
7
8  public class SubprogramaSuma extends JApplet {
9     double suma; // la suma de los valores introducidos por el usuario
10
11    // inicializar el applet, obteniendo los valores del usuario
12    public void init()
13    {
14       String primerNumero;    // primera cadena introducida por el usuario
15       String segundoNumero;   // segunda cadena introducida por el usuario
16
17       double numero1;         // primer número a sumar
18       double numero2;         // segundo número a sumar
19
20       // obtener el primer número del usuario
21       primerNumero = JOptionPane.showInputDialog(
22          "Escriba el primer valor de punto flotante" );
23
24       // obtener el segundo número del usuario
25       segundoNumero = JOptionPane.showInputDialog(
26          "Escriba el segundo valor de punto flotante" );
27
28       // convertir los números de tipo String a tipo double
29       numero1 = Double.parseDouble( primerNumero );
30       numero2 = Double.parseDouble( segundoNumero );
31
32       // sumar los números
33       suma = numero1 + numero2;
34
35    } // fin del método init
36
37    // dibujar los resultados en un rectángulo en el fondo del applet
38    public void paint( Graphics g )
39    {
40       // llamar a la versión del método paint de la superclase
41       super.paint( g );
42
43       // dibujar un rectángulo, empezando desde (15, 10), que tenga 270
44       // píxeles de ancho y 20 píxeles de alto
45       g.drawRect( 15, 10, 270, 20 );
46
47       // dibujar los resultados como String en (25, 25)
48       g.drawString( "La suma es " + suma, 25, 25 );
49
50    } // fin del método paint
51
52 } // fin de la clase SubprogramaSuma
```

Figura 3.13 Sumar valores **double**. (Parte 1 de 2.)

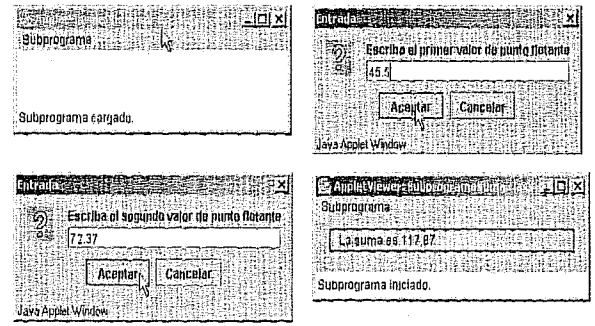


Figura 3.13 Sumar valores **double**. (Parte 2 de 2.)

Una vez más, utilizamos **JOptionPane.showInputDialog** para solicitar datos de entrada al usuario. El applet calcula la suma de los valores de entrada y muestra el resultado, dibujando una cadena dentro de un rectángulo en el applet. El documento HTML para cargar este applet en el **appletviewer** se muestra en la figura 3.14.

La línea 5 de la figura 3.13

```
import java.awt.Graphics;    // importar la clase Graphics
```

importa la clase **Graphics** (del paquete **java.awt**) para utilizarla en este applet. En realidad no se requiere la declaración **import** de la línea 5, si siempre utilizamos el nombre completo de la clase **Graphics** (***java.awt.Graphics***), que incluye el nombre completo del paquete y el nombre de la clase. Por ejemplo, la primera línea del método **paint** podría escribirse así:

```
public void paint( java.awt.Graphics g )
```

Observación de ingeniería de software 3.1

El compilador de Java no requiere las declaraciones **import** *en un archivo de código fuente en Java, si se especifica el nombre completo de la clase (también conocido como el nombre de clase completamente calificado) cada vez que se utilice el nombre de esa clase en el código fuente. Por ejemplo, el nombre de clase completamente calificado para la clase* **Graphics** *es* **java.awt.Graphics**.

La línea 6

```
import javax.swing.* // importar el paquete javax.swing
```

especifica al compilador que todas las clases del paquete **javax.swing** están disponibles para este applet. El asterisco (*) indica que todas las clases en el paquete **javax.swing** (como **JApplet** y **JOptionPane**) deben estar disponibles para el compilador. Sin embargo, esto no hace que todas las clases del paquete se carguen en memoria cuando el programa empieza a ejecutarse; sólo se cargarán aquellas clases que se utilicen en el programa. Esto le permite a usted utilizar el *nombre simple* (el nombre de la clase en sí) de cualquier clase del paquete **javax.swing**, en el programa. Nuestros últimos dos programas importaron solamente la clase **JApplet** del paquete **javax.swing**. En este programa utilizamos las clases **JApplet** y **JOptionPane** de ese paquete.

```
1  <html>
2  <applet code = " applet Suma.class" width = "300" height = "65">
3  </applet>
4  </html>
```

Figura 3.14 **SubprogramaSuma.html** carga la clase **SubprogramaSuma** de la figura 3.13 en el **appletviewer**.

http://librosysolucionarios.net

Observación de ingeniería de software 3.2

Muchos paquetes tienen subpaquetes. Por ejemplo, el paquete `java.awt` *tiene el subpaquete* `java.awt.event`. *Cuando el compilador encuentra una declaración* `import` *que utiliza la notación* `*` *(por ejemplo,* `java.awt.*`*) para indicar que un programa podría utilizar varias clases de un paquete, el compilador no carga las clases de ninguno de los subpaquetes de* `java.awt`. *Por lo tanto, no se puede definir una instrucción* `import` *con* `java.*` *para buscar las clases de todos los paquetes básicos de Java.*

Error común de programación 3.4

Asumir que una declaración `import`, *en un paquete completo, también importa clases de los subpaquetes de ese paquete, produce un error de sintaxis para las clases de esos subpaquetes.*

Recuerde de la sección 3.3 que los applets heredan de la clase `JApplet`, por lo que tienen todos los métodos que requiere un contenedor de applets para ejecutarlos. La línea 8

```
public class SubprogramaSuma extends JApplet {
```

comienza la declaración de la clase `SubprogramaSuma`, e indica que esta clase hereda de `JApplet`. El cuerpo de la clase está delimitado por las llaves en las líneas 8 y 52.

La línea 9

```
double suma;  // la suma de los valores introducidos por el usuario
```

declara que `suma` es una variable del tipo primitivo `double`. Como la variable está declarada dentro del cuerpo de la declaración de la clase, pero fuera de los cuerpos de todas las declaraciones de los métodos de la clase, es una *instrucción de declaración de variable* para un *campo* (lo que algunas veces se conoce como *variable de instancia* en otros lenguajes de programación). Cada instancia de la clase contiene una copia de cada campo. Por ejemplo, si hay 10 objetos `SubprogramaSuma`, cada objeto tiene su propia copia de `suma`. Por lo tanto, habría 10 copias de `suma` (una por cada applet). [*Nota:* Utilizamos el término de Java "campo", en vez del término "variable de instancia", a lo largo del libro.]

Un beneficio de los campos es que todos los métodos de la clase pueden usarlos. Hasta ahora, hemos declarado todas las variables en el método `main` de una aplicación. Las variables que se declaran en el cuerpo de un método específico se conocen como *variables locales*, y pueden utilizarse solamente en el cuerpo de ese método. Otra de las distinciones entre los campos y las variables locales es que los campos tienen *valores predeterminados* y las variables locales no. El valor predeterminado para un campo de tipo `double` (`suma` en este ejemplo) es 0.0.

Buena práctica de programación 3.2

Inicializar en forma explícita los campos, en vez de depender de una inicialización automática, mejora la legibilidad de los programas.

El applet de la figura 3.13 contiene dos métodos: `init` (líneas 12 a 35) y `paint` (líneas 38 a 50). Cuando un contenedor de applets carga un applet, el contenedor crea una instancia de la clase del applet y llama a su método `init`. El contenedor de applets llama al método `init` sólo una vez, durante la ejecución de un applet. El método `init` normalmente *inicializa* los campos del applet (si necesitan inicializarse con valores distintos a los predeterminados) y realiza tareas que deben ocurrir sólo una vez, cuando el applet comienza a ejecutarse. Como veremos en capítulos posteriores, el método `init` del applet generalmente crea su interfaz gráfica de usuario.

Observación de ingeniería de software 3.3

El orden en el que los métodos se declaran en una clase, no determina cuándo se llaman esos métodos en tiempo de ejecución. Sin embargo, las convenciones en cuanto al orden en el que los métodos se declaran, mejoran la legibilidad y facilidad de mantenimiento del programa.

La primera línea del método `init` siempre aparece como

```
public void init()
```

indicando que `init` es un método `public` que no recibe argumentos (paréntesis vacíos después de `init`) para realizar su tarea, y no devuelve información (`void`) al terminar. El cuerpo del método `init` está definido por el par de llaves en las líneas 13 y 35.

Las líneas 14 y 15

```
String primerNumero;   // primera cadena introducida por el usuario
String segundoNumero;  // segunda cadena introducida por el usuario
```

declaran las variables `String` `primerNumero` y `segundoNumero`, en las que el applet guarda las cadenas introducidas por el usuario. Estas variables se declaran en el cuerpo del método `init`, por lo que son variables locales de este método.

Las líneas 17 y 18

```
double numero1;        // primer número a sumar
double numero2;        // segundo número a sumar
```

declaran las variables `numero1` y `numero2` de tipo primitivo `double` (estas variables guardan valores de punto flotante). A diferencia del campo `suma`, las variables locales `numero1` y `numero2` no se inicializan con 0.0.

Analizaremos más las diferencias entre las declaraciones `String` en las líneas 14 y 15, y las declaraciones `double` en las líneas 17 y 18. Los tipos en Java se dividen en dos categorías: *tipos primitivos* y *tipos de referencia* (conocidos también como *tipos no primitivos*). Los tipos primitivos son `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`. Las variables de esos tipos pueden, cada una, guardar exactamente un valor de su tipo declarado. Por ejemplo, la variable `numero1` puede guardar exactamente un valor `double` (punto flotante) a la vez. Todos los tipos no primitivos son tipos de referencia, por lo que todas las clases[16] son tipos de referencia. Los programas utilizan variables de tipos de referencia (generalmente conocidos como *referencias*) para *referirse a los objetos* en el programa. Dichas referencias contienen la ubicación de un objeto en la memoria de la computadora, el cual podría contener muchas piezas de datos, y podría tener muchos métodos. Un `String` es un objeto, y las variables `primerNumero` y `segundoNumero` (líneas 14 y 15) son realmente referencias a objetos `String`.

Una variable de tipo primitivo no puede usarse para invocar a un método. Sin embargo, un uso importante de una referencia es para invocar métodos de cierto objeto. En nuestros applets anteriores, el método `paint` recibe una referencia llamada `g`, la cual hace referencia a un objeto `Graphics`. Las instrucciones en el método `paint` utilizan esa referencia para enviar mensajes al objeto `Graphics`. Estos mensajes son llamados a métodos (como `drawString` y `drawLine`), los cuales hacen que el programa dibuje. Por ejemplo, la instrucción

```
g.drawString( "¡Bienvenido a la programación en Java!", 25, 25 );
```

utiliza a `g` para enviar el mensaje `drawString` a (es decir, llamar al método `drawString`) el objeto `Graphics`. Como parte del mensaje (llamada a método), proporcionamos la información (argumentos) que `drawString` requiere para realizar su tarea. El objeto `Graphics` utiliza esta información para dibujar la cadena en la ubicación especificada.

Observación de ingeniería de software 3.4

Una sugerencia para ayudarle a determinar si una variable es de tipo primitivo o de referencia, es el nombre del tipo declarado de esa variable. Por convención, todos los nombres de clases en Java empiezan con letra mayúscula. Por lo tanto, si el tipo empieza con mayúscula, generalmente se puede asumir que la variable es una referencia a un objeto del tipo declarado (por ejemplo, `Graphics g` *indica que* `g` *es una referencia a un objeto* `Graphics`*).*

Las líneas 21 y 22

```
primerNumero = JOptionPane.showInputDialog(
   "Escriba el primer valor de punto flotante" );
```

leen el primer número de punto flotante introducido por el usuario. El método `showInputDialog` de `JOptionPane` muestra un cuadro de diálogo de entrada que pide al usuario que escriba un valor. El usuario escribe un valor en el campo de texto del cuadro de diálogo de entrada y después hace clic en el botón **Aceptar** para regresar la cadena que escribió el usuario. Si usted escribe y no aparece nada en el campo de texto, coloque

16. En el capítulo 9, hablaremos sobre las interfaces, que también son tipos de referencias.

http://librosysolucionarios.net

el puntero del ratón en el campo de texto y haga clic para activarlo. A la variable `primerNumero` se le asigna el resultado de la llamada a `showInputDialog`.

En las líneas 21 y 22, observe la sintaxis de la llamada al método. Hasta este momento, hemos visto dos llamadas distintas a los métodos de llamada. Esta instrucción utiliza la sintaxis de llamada a métodos `static`, presentada en el capítulo 2. Recuerde que los métodos `static` se llaman con la sintaxis

NombreClase.nombreMétodo(argumentos)

En este capítulo también hemos llamado métodos de la clase `Graphics` con una sintaxis similar que empieza con una referencia a un objeto `Graphics`. En forma genérica, esta sintaxis es

nombreReferencia.nombreMétodo(argumentos)

Esta sintaxis se utiliza para la mayoría de las llamadas a métodos en Java. De hecho, el contenedor de applets utiliza esta sintaxis para llamar a los métodos `init`, `start` y `paint` en los applets que usted ejecute.

Las líneas 25 y 26

```
segundoNumero = JOptionPane.showInputDialog(
   "Escriba el segundo valor de punto flotante" );
```

leen el segundo número de punto flotante introducido por el usuario mediante un cuadro de diálogo.

Las líneas 29 y 30

```
numero1 = Double.parseDouble( primerNumero );
numero2 = Double.parseDouble( segundoNumero );
```

convierten las dos cadenas introducidas por el usuario en valores `double`, para usarlos en un cálculo. El método `Double.parseDouble` (un método `static` de la clase `Double`) convierte su argumento `String` en valor `double` de punto flotante. La clase `Double` se encuentra en el paquete `java.lang`.



Observación de ingeniería de software 3.5

Cada tipo primitivo (como `double`) tiene una clase correspondiente (como `Double`) en el paquete `java.lang`. Estas clases [conocidas como clases de envoltura de tipos *(wrapper)] proporcionan métodos para procesar valores de tipo primitivo (como convertir una cadena a un valor de tipo primitivo, o viceversa). Los tipos primitivos no tienen métodos, por lo que los métodos relacionados con un tipo primitivo se localizan en la clase de envoltura de tipos correspondientes (por ejemplo, el método `parseDouble`, que convierte un valor `String` en `double`, se localiza en la clase `Double`). Consulte en la documentación del API los detalles completos sobre los métodos en las clases de envoltura de tipos.*

La instrucción de asignación en la línea 33

```
suma = numero1 + numero2;
```

calcula la suma de los valores almacenados en las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma`, utilizando el operador de asignación `=`. La instrucción se lee como "`suma` *recibe* el valor de `numero1 + numero2`". Observe que el campo `suma` se utiliza en el método `init`, incluso aunque `suma` no se definió en el método `init`. Podemos usar `suma` en `init` (y en todos los demás métodos de la clase), ya que `suma` es un campo.

En este punto, el método `init` del applet regresa y el contenedor del applet llama al método `start` del applet. No declaramos `start` en este applet, por lo que se llama al que se hereda de la clase `JApplet`. Generalmente, el método `start` se utiliza con un concepto avanzado, conocido como subprocesamiento múltiple.[17]

A continuación, el contenedor de applets llama al método `paint` del applet. En este ejemplo, el método `paint` dibuja un rectángulo en el que aparecerá el resultado de la suma. La línea 45

```
g.drawRect( 15, 10, 270, 20 );
```

llama al método `drawRect` de la clase `Graphics`, utilizando la referencia `g`. El método `drawRect` dibuja un rectángulo con base en sus cuatro argumentos. Los primeros dos valores enteros representan la *coordenada x superior izquierda* y la *coordenada y superior izquierda* en donde el objeto `Graphics` comienza a dibujar el rectángulo. Los argumentos tercero y cuarto son enteros no negativos que representan la anchura y la altura del rectángulo en píxeles, respectivamente. Esta instrucción específica dibuja un rectángulo que empieza en la

17. Veremos los usos típicos del método `start` en los capítulos 16 y 19.

http://librosysolucionarios.net

coordenada *(15, 10)*, que tiene 270 píxeles de ancho y 20 píxeles de altura. Hay que tener en cuenta que experimentamos con estos argumentos hasta encontrar valores que colocaran lo mejor posible al rectángulo alrededor de la cadena en el applet.



Error común de programación 3.5

Es un error lógico proporcionar dos puntos (es decir, pares de coordenadas x e y) como argumentos para el método `drawRect` de `Graphics`. El tercer argumento debe ser la anchura en píxeles, y el cuarto argumento debe ser la altura en píxeles del rectángulo a dibujar.



Error común de programación 3.6

Es un error lógico proporcionar una anchura o altura negativa como argumento para el método `drawRect` de la clase `Graphics`. El rectángulo no aparecerá y no se indicará ningún error.



Error común de programación 3.7

Por lo general, es un error lógico proporcionar argumentos al método `drawRect` de `Graphics` que ocasionen que el rectángulo se dibuje fuera del área visible del applet (es decir, la anchura y altura del applet, según lo especificado en el documento HTML que hace referencia al applet). Debe incrementar la anchura o la altura del applet en el documento HTML, o utilizar argumentos para el método `drawRect` que hagan que el rectángulo se dibuje dentro del área visible del applet.

La línea 48

```
g.drawString( "La suma es " + suma, 25, 25 );
```

llama al método `drawString` del objeto `Graphics`. La expresión

```
"La suma es "  + suma
```

utiliza el operador de concatenación de cadenas `+` para concatenar la cadena "`La suma es`" y `suma` (convertida en cadena) para crear la cadena que `drawString` va a mostrar. Observe de nuevo que la instrucción utiliza el campo `suma`, incluso aunque el método `paint` no define a `suma` como una variable local.

El beneficio de declarar `suma` como un campo es que podemos asignar a `suma` un valor en `init`, y podemos utilizar más adelante el valor de `suma` en el método `paint` en el programa. Todos los métodos de una clase son capaces de utilizar los campos de esa clase.[18]



Observación de ingeniería de software 3.6

Las únicas instrucciones que deben colocarse en el método `init` de un applet son las que están directamente relacionadas con la primera inicialización de los campos de un applet. Los resultados del applet deben mostrarse desde los demás métodos de la clase del applet.



Observación de ingeniería de software 3.7

Las únicas instrucciones que deben colocarse en el método `paint` de un applet son aquellas que están directamente relacionadas con el dibujo (es decir, las llamadas a los métodos de la clase `Graphics`) y la lógica del dibujo. En general, los cuadros de diálogo no deben mostrarse desde el método `paint` de un applet.

En este capítulo y en el capítulo 2, presentamos y reforzamos muchos conceptos importantes de Java como aplicaciones, applets, mostrar datos en la pantalla, introducir datos mediante el teclado, realizar cálculos y tomar decisiones. En el capítulo 4 veremos más sobre esas técnicas, cuando introduzcamos la *programación estructurada*. También estudiaremos cómo especificar y variar el orden en el que un programa ejecuta las instrucciones; este orden se conoce como *flujo del control*.

3.6 Recursos en Internet y Web relacionados con los applets de Java

Si usted tiene acceso a Internet y World Wide Web, existe una gran cantidad de recursos para applets de Java disponibles para usted. El mejor lugar para empezar es en el origen: el sitio Web sobre Java de Sun Microsystems, Inc., `java.sun.com`. La página Web

```
java.sun.com/applets
```

18. En el capítulo 8 veremos una excepción a este enunciado.

contiene varios recursos sobre applets de Java, incluyendo applets gratuitos que usted puede utilizar en su propio sitio Web, los applets de demostración del J2SDK y otros applets (muchos de los cuales puede descargar y utilizar en su propio equipo). También hay una sección titulada "Applets at Work", en donde puede leer acerca de los usos de los applets en la industria.

Si no tiene el Complemento Java (Java Plug-in) en su equipo, puede visitar

`java.sun.com/getjava` (versión en inglés) o `java.sun.com/getjava/es` (versión en español)

para descargar e instalar el complemento. Se incluyen instrucciones para varias versiones de Windows, Linux, Solaris y Mac OS.

En el sitio Web sobre Java de Sun Microsystems, visite el sitio *Java Developer Services*

`developer.java.sun.com`

Este sitio gratuito incluye soporte técnico, foros de discusión, artículos técnicos, recursos, anuncios de nuevas características de Java y acceso anticipado a las nuevas tecnologías de Java. Sólo tenga en cuenta que mucho del material mencionado aparece en inglés.

Para ver los diversos cursos de capacitación y tutoriales en línea, visite el sitio

`developer.java.sun.com/developer/onlineTraining`

Otro sitio Web útil es *JARS*: originalmente conocido como el *Servicio de Clasificación de applets en Java (Java Applet Rating Service)*.

El sitio JARS

`www.jars.com`

originalmente era un extenso depósito para applets de Java. Su beneficio era que clasificaba cada uno de los applets registrados en el sitio, de manera que se encontraban entre el 1, el 5, o el 25% de los más populares. De esta forma usted podía ver los mejores applets en Web. Durante las primeras etapas del desarrollo del lenguaje Java, hacer que calificaran el applet de un programador en este sitio, era una excelente manera de demostrar sus habilidades de programación en Java. JARS es ahora un recurso completo para los programadores de Java.

Los recursos que se enlistan en esta sección proporcionan hipervínculos para muchos otros sitios Web relacionados con Java. Si usted tiene acceso a Internet, invierta más tiempo navegando por estos sitios, ejecutando applets y leyendo el código fuente de éstos, cuando esté disponible. Esto le ayudará a expandir rápidamente su experiencia con Java. Proporcionamos muchos otros recursos sobre Java basados en Web con los recursos para *Cómo programar en Java, quinta edición*, en `www.deitel.com`.

3.7 (Ejemplo práctico opcional) Acerca de los objetos: Identificación de las clases en el enunciado de un problema

Ahora comenzaremos la considerable tarea de diseñar el simulador de elevador que presentamos en el capítulo 2. Empezaremos con el diseño del elevador en sí, y diseñaremos la interacción del usuario y el despliegue de este modelo en las secciones 14.13 y 19.7, respectivamente.

Identificación de las clases en un sistema

El primer paso de nuestro proceso de DOO es identificar los objetos a partir del enunciado del problema, para agruparlos en clases. Eventualmente describiremos estas clases de una manera formal, y las implementaremos en Java. Primero revisaremos el enunciado del problema de la sección 2.9, para identificar los *sustantivos* clave; es probable que algunos de estos sustantivos identifiquen los objetos que conforman la simulación del elevador.[19] La figura 3.15 muestra estos sustantivos (y frases nominales).

Seleccionamos sólo los sustantivos que tengan significado en la simulación del elevador. Por esta razón omitimos varios sustantivos. No necesitamos modelar "empresa" como una clase, ya que la empresa no forma parte de la simulación; la empresa simplemente quiere que nosotros construyamos la simulación del elevador. No vamos a modelar el "edificio de oficinas", la ubicación del elevador, ya que el edificio se encuentra fuera del alcance de nuestra simulación del elevador. Las frases "pantalla", "audio" y "música de elevador" pertenecen a la pantalla de la simulación, pero no pertenecen a la simulación en sí. (Utilizaremos estas frases cuando construyamos la pantalla en la sección 19.7 y en el apéndice F.) Las frases "interfaz gráfica de usuario (GUI)", "usuario de nuestra aplicación", "botón de GUI **Primer Piso**" y "botón de GUI **Segundo Piso**" pertenecen a la manera en que el usuario controla el simulador. (Utilizaremos estas frases cuando construyamos la interfaz de usuario en la sección 14.13.) Aunque ahorraremos energía con la política de no mover el elevador sino hasta que se solicite, no modelamos la "energía". Por último, "capacidad" es una propiedad del elevador y del piso, no una entidad por separado.

19. El análisis de los sustantivos es un simple medio por el cual podemos determinar los objetos requeridos para implementar el problema. Como dijimos en el capítulo 1, en la industria se utilizan a menudo procesos más complejos.

http://librosysolucionarios.net

Sustantivos (y frases nominales) en el enunciado del problema

empresa	sistema de elevador	interfaz gráfica de usuario (GUI)
edificio de oficinas	conducto del elevador	persona
elevador	puerta del piso	piso (primer piso, segundo piso)
aplicación de simulador de software	timbre dentro del elevador	botón de GUI **Primer Piso**
pasajero	luz en ese piso	botón de GUI **Segundo Piso**
usuario de nuestra aplicación	energía	audio
botón del piso	capacidad	música de elevador
botón del elevador	carro del elevador	pantalla

Figura 3.15 Los sustantivos (y frases nominales) en el enunciado de un problema.

Para determinar las clases de nuestro sistema, agrupamos los sustantivos restantes. Descartaremos "sistema de elevador" en estos momentos; nos enfocaremos en diseñar sólo el simulador, sin tomar en cuenta cómo se relaciona este simulador con el sistema en su totalidad. (Hablaremos sobre el sistema en su totalidad en la sección 14.13.) Combinaremos "elevador" y "carro del elevador" en "elevador", ya que el enunciado del problema utiliza las dos palabras para describir lo mismo. Cada sustantivo restante de la figura 3.15 se refiere a uno o más de los siguientes:

- conducto del elevador
- elevador
- persona
- piso (primer piso, segundo piso)
- puerta del elevador
- puerta del piso
- botón del elevador
- botón del piso
- timbre
- luz

Es probable que los elementos de esta lista vayan a ser clases que necesitaremos implementar en nuestro sistema. Observe que enlistamos los botones en los pisos y el botón en el elevador por separado. Los dos tipos de botones realizan distintas tareas en nuestra simulación: los botones en los pisos llaman al elevador y el botón en el elevador lo instruye a moverse hacia el otro piso.

Ahora podemos modelar las clases en nuestro sistema, con base en la lista que creamos. Por convención, capitalizamos los nombres de las clases en el proceso de diseño (como lo haremos cuando escribamos el código Java que implementa nuestro diseño.) Si el nombre de una clase contiene más de una palabra, juntamos las palabras y capitalizamos cada una (por ejemplo, `NombreConVariasPalabras`). Utilizando esta convención, creamos las clases `ConductoElevador`, `Elevador`, `Persona`, `Piso`, `PuertaElevador`, `PuertaPiso`, `BotonElevador`, `BotonPiso`, `Timbre` y `Luz`. Crearemos nuestro sistema utilizando todas estas clases como bloques de construcción. Antes de empezar a construir el sistema, debemos comprender mejor la manera en que se relacionan las clases entre sí.

Cómo modelar las clases

UML nos permite modelar, mediante el *diagrama de clases*, las clases en el sistema del elevador y sus relaciones. Los diagramas de clases modelan la estructura de un sistema mediante la descripción de sus clases o "bloques de construcción". La figura 3.16 representa la clase `Elevador`. En UML, cada clase se modela como un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase. El compartimiento de en medio contiene los *atributos* (hablaremos sobre los atributos en las secciones 4.14 y 5.11). El compartimiento inferior contiene las *operaciones* (las cuales se describen en la sección 6.15).

La figura 3.17 muestra cómo se relacionan nuestras clases `ConductoElevador`, `Elevador` y `BotonPiso` entre sí. Observe que los rectángulos en este diagrama no se subdividen en compartimientos. UML permite suprimir los atributos y las operaciones de las clases en esta forma, para crear diagramas más legibles. Se dice que un diagrama de este tipo es un *diagrama con elementos omitidos (elided diagram)*; un diagrama en el que cierta información, como el contenido de los compartimientos segundo y tercero, no se modela. Colocaremos información en estos compartimentos en las secciones 4.14 y 6.15, respectivamente.

En la figura 3.17, una línea sólida que conecta las clases representa una *asociación*. Una asociación es una relación entre clases. Los números cerca de las líneas expresan los valores de *multiplicidad*, que indican cuántos objetos de una clase participan en la asociación. Mediante el diagrama podemos ver que dos objetos de la clase `BotonPiso` participan en una asociación con un objeto de la clase `ConductoElevador`, ya que los dos botones de piso están localizados en el conducto del elevador. Por lo tanto, la clase `BotonPiso` tiene una relación de *dos a uno* con la clase `ConductoElevador`; la clase `ConductoElevador` tiene una relación de *uno a dos* con la clase `BotonPiso`. También podemos ver que la clase `ConductoElevador` tiene una relación de *uno a uno* con la clase `Elevador` y viceversa. Utilizando UML podemos modelar muchos tipos de multiplicidad. La figura 3.18 muestra los tipos de multiplicidad y cómo representarlos.

Una asociación puede tener un nombre. Por ejemplo, la palabra `Solicita` arriba de la línea que conecta a las clases `BotonPiso` y `Elevador` indica el nombre de esa asociación; la flecha muestra la dirección del nombre de la asociación. Esta parte del diagrama se lee "dos objetos de la clase `BotonPiso` solicitan un objeto de la clase `Elevador`". Observe que los nombres de asociaciones van en la misma dirección que indica la punta de flecha que va enseguida del nombre de la asociación; por lo tanto, sería impropio leer, por ejemplo, la asociación anterior como "un objeto de la clase `Elevador` solicita dos objetos de la clase `BotonPiso`". Además, la palabra `Restablece` indica que "un objeto de la clase `ConductoElevador` restablece a dos objetos de la clase `BotonPiso`". Por último, la frase `Indica la llegada` muestra que "un objeto de la clase `Elevador` indica la llegada del objeto `Elevador` a un objeto de la clase `ConductoElevador`".

Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más complejas en donde los objetos de una clase están compuestos por objetos de otras clases. Por ejemplo, considere el caso de



Figura 3.16 Representación de una clase en UML.

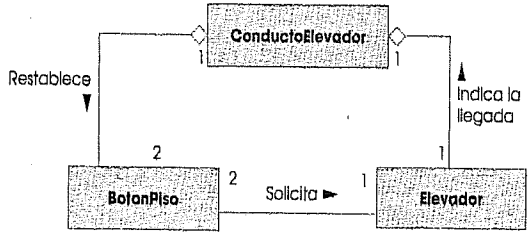


Figura 3.17 Diagrama de clases que muestra las asociaciones entre las clases.

http://librosysolucionarios.net

Símbolo	Significado
0	Ninguno
1	Uno
m	Un valor entero
0..1	Cero o uno
m, n	*m* o *n*
m..n	Cuando menos *m*, pero no menor que *n*
*	Cero o más
0..*	Cero o más
1..*	Uno o más

Figura 3.18 Tipos de multiplicidad.

un elevador real. ¿Qué "piezas" reúne un fabricante para construir un elevador funcional? La empresa que nos pidió construir esta simulación nos dice que este sistema completo de elevador está compuesto de un conducto de elevador, dos botones de piso, dos luces, dos puertas de piso y un carro elevador. El carro elevador en sí está compuesto de varias piezas, incluyendo un botón de elevador, una puerta de elevador y un timbre. A partir de esta descripción, podemos ver que un elevador funcional es algo complejo, construido a partir de muchas piezas.

Los *diamantes*[20] en las líneas de las asociaciones de la figura 3.17 indican que `ConductoElevador` está construido a partir de otras piezas. La relación denotada por esta asociación se llama *agregación*: un `ConductoElevador` se crea al agregar, o reunir, varios objetos más para producir todo un `ConductoElevador`. La agregación implica una relación entre un todo/una parte. La clase que tiene el símbolo de agregación en su extremo de una línea de asociación es el todo (en este caso, `ConductoElevador`), y la clase al otro extremo de la línea de la asociación es la parte (en este caso, las clases `BotonPiso` y `Elevador`). En este diagrama, la agregación indica que un objeto de la clase `ConductoElevador` se compone de dos objetos de la clase `BotonPiso` y un objeto de la clase `Elevador`. El conducto del elevador "tiene un" elevador y dos botones de piso. La relación "tiene un" define a la agregación.

La figura 3.19 muestra el diagrama de clases completo para el modelo del elevador, con base en las relaciones entre los objetos que se describieron en el enunciado del problema. Modelamos todas las clases que identificamos, junto con las asociaciones entre estas clases. [*Nota:* En el capítulo 10 expandiremos nuestro diagrama de clases, utilizando el concepto orientado a objetos de la herencia.]

La clase `ConductoElevador` es una agregación de un objeto de la clase `Elevador` y dos objetos de cada una de las clases `Luz`, `PuertaPiso` y `BotonPiso`. (Observe las relaciones de dos a uno entre cada una de estas clases y `ConductoElevador`.) La clase `Elevador` es una agregación de las clases `PuertaElevador`, `BotonElevador` y `Timbre`. La clase `Persona` tiene asociaciones con `BotonPiso` y `BotonElevador` (y con otras clases, como veremos pronto). El nombre de asociación `Oprime` y las flechas de dirección de ese nombre indican que el objeto de la clase `Persona` oprime estos botones. El objeto de la clase `Persona` también sube al objeto de la clase `Elevador` y camina a lo largo del objeto de la clase `Piso`. El nombre de asociación `Solicita` indica que un objeto de la clase `BotonPiso` solicita un objeto de la clase `Elevador`. El nombre de asociación `Indica que se mueva` señala que el objeto de la clase `BotonElevador` indica al objeto de la clase `Elevador` que se mueva al otro piso. El diagrama muestra también muchas otras asociaciones.

Observe la palabra `pasajero` en la línea de la asociación entre `Persona` y `Elevador`. Éste es un *nombre de rol*, el cual identifica el rol que el objeto `Persona` desempeña en su relación con el `Elevador`. Un nombre de rol añade significado a una asociación entre clases. En este caso, el nombre de rol `pasajero` indica

20. Los diamantes huecos en estos diagramas de clases indican una agregación simple. Los diamantes sólidos indicarían una forma más fuerte de agregación, conocida como composición o agregación compuesta. Hablaremos sobre la composición y la compararemos con la asociación simple en el capítulo 8.

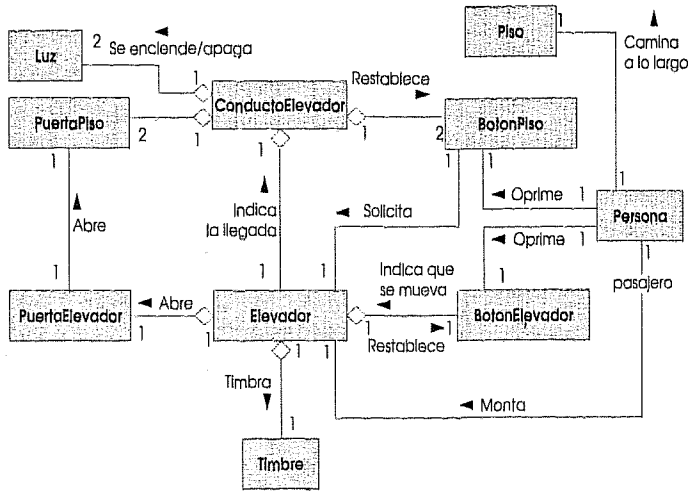


Figura 3.19 Diagrama de clases para el modelo del elevador.

que, cuando una `Persona` sube al `Elevador`, esa persona es un pasajero del `Elevador`. Para muchas de las asociaciones en este diagrama de clases los nombres de rol explícitos no son necesarios, ya que los significados de las asociaciones son claros sin ellos.

Ahora hemos identificado las clases para nuestro sistema (aunque tal vez descubramos otras en las siguientes fases del proceso de diseño). En la sección 4.14 determinaremos los atributos para cada una de estas clases, y en la sección 5.11 utilizaremos estos atributos para examinar cómo cambia el sistema con el tiempo, y para presentar sus aspectos del comportamiento. A medida que aumentemos nuestro conocimiento, descubriremos nueva información que nos permitirá describir de manera más detallada a nuestras clases.

Preguntas

1. ¿Por qué podría ser más complicado implementar un edificio de tres pisos (o más)?
2. Es común que los edificios grandes tengan muchos elevadores. En el capítulo 10 veremos que, una vez que hayamos creado un objeto elevador, es fácil crear todos los que queramos. ¿Qué problemas u oportunidades anticipa usted en cuanto a tener varios elevadores, cada uno de los cuales podría recoger y despachar pasajeros en cualquier piso, en un edificio grande?
3. Por cuestiones de simpleza, hemos dado a nuestro elevador y a cada piso la capacidad de un pasajero. ¿Qué problemas y oportunidades anticipa usted en cuanto a tener que aumentar estas capacidades?

RESUMEN

- Los applets, son programas en Java que pueden incrustarse en documentos HTML (HyperText Markup Language, Lenguaje de Marcado de Hipertexto) (es decir, páginas Web). Cuando un navegador carga una página Web que contiene un applet, éste se descarga en el navegador Web y comienza a ejecutarse.
- En el `appletviewer` usted puede ejecutar un applet otra vez, haciendo clic en el menú **Subprograma (Applet)** del `appletviewer` y seleccionando el elemento de menú **Volver a cargar (Reload)**. Para terminar un applet, seleccione el elemento de menú **Salir (Quit)**.
- La clase `Graphics` se encuentra en el paquete `java.awt`.
- La clase `JApplet` se encuentra en el paquete `javax.swing`. Los contenedores de applets que soportan `JApplet` esperan que cada applet de Java tenga ciertos comportamientos que `JApplet` proporciona de antemano.

- La herencia se utiliza para crear nuevas clases, a partir de las declaraciones de clases existentes. La palabra clave `extends`, seguida de un nombre de clase, indica la superclase de la cual una nueva clase hereda. La nueva clase es la subclase. La herencia produce una subclase que tiene los atributos y comportamientos de la superclase, así como las nuevas características agregadas en la declaración de la subclase.
- Las clases se utilizan como "plantillas" o "planos de construcción" para instanciar objetos en memoria y utilizarlos en un programa. El término objeto normalmente implica que los atributos y comportamientos están asociados con el objeto, y que esos comportamientos realizan operaciones sobre los atributos.
- El método `paint` es uno de tres métodos que son llamados por un contenedor de applets cuando un applet empieza a ejecutarse. Esos métodos son `init`, `start` y `paint`, y se llaman en ese orden.
- Los métodos declaran los datos requeridos para completar sus tareas mediante sus listas de parámetros. Generalmente, los programadores pasan los datos al llamar a los métodos. Para el método `paint`, el contenedor de applets llama al método y le pasa el argumento `Graphics`.
- El método `drawString` de la clase `Graphics` dibuja una cadena en una ubicación especificada. Su primer argumento es la cadena a dibujar. Los últimos dos argumentos son las coordenadas de la esquina inferior izquierda de la cadena. Las coordenadas se miden en píxeles, a partir de la coordenada superior izquierda *(0, 0)* de un applet.
- Se debe crear un documento HTML para cargar un applet en un contenedor de applets. Por lo general, el applet y su correspondiente documento HTML se guardan en el mismo directorio.
- Muchos elementos del HTML tienen pares de etiquetas (tags). Cada etiqueta va encerrada entre los signos < y >.
- El primer componente de la etiqueta `<applet>` indica el archivo que contiene la clase del applet compilada. Los componentes segundo y tercero de la etiqueta `<applet>` indican la anchura (`width`) y la altura (`height`) del applet, en píxeles. El `appletviewer` sólo entiende las etiquetas `<applet>` y `</applet>` de HTML, por lo que algunas veces se conoce como el "navegador mínimo".
- Las declaraciones `import` no se requieren si usted utiliza siempre el nombre completamente calificado de una clase, que incluye el nombre del paquete y de la clase (por ejemplo, `java.awt.Graphics`).
- La notación del asterisco (`*`) después del nombre de un paquete en una declaración `import` indica que pueden utilizarse varias clases de ese paquete en el programa. Esto permite a los programadores utilizar el nombre simple (el nombre de la clase por sí solo) de cualquier clase del paquete en el programa.
- Todo objeto (instancia de una clase) contiene una copia de cada uno de los campos de esa clase. Los campos se declaran en el cuerpo de la declaración de una clase, pero no en el cuerpo de los métodos. Los campos pueden utilizarse en todos los métodos de la clase.
- Las variables que se declaran en el cuerpo de un método son variables locales; pueden utilizarse solamente en el cuerpo del método en el que se declaran.
- A los campos siempre se les asigna un valor predeterminado, y a las variables locales no.
- El método `init` normalmente inicializa los campos de un applet y realiza cualquier tarea que deba ocurrir sólo una vez, cuando el applet empiece a ejecutarse.
- Los tipos en Java se dividen en dos categorías: tipos primitivos y tipos de referencia (algunas veces conocidos como tipos no primitivos). Los tipos primitivos son `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`. Cada una de las variables de esos tipos puede almacenar exactamente un valor de su tipo declarado. Una variable de tipo primitivo no puede utilizarse como referencia para invocar a un método.
- Todos los tipos no primitivos son tipos de referencia, por lo cual todas las clases son tipos de referencia. Los programas utilizan variables de tipos de referencia (normalmente conocidas como referencias) para hacer referencia a los objetos en el programa. Dichas referencias contienen la ubicación de un objeto en la memoria de la computadora, y un objeto podría contener muchas piezas de datos y muchos métodos. Un uso importante de una referencia es para invocar métodos de un objeto particular.
- El tipo primitivo `double` almacena números de punto flotante, con precisión doble. El tipo primitivo `float` almacena números de punto flotante, con precisión simple. Un `double` requiere de más memoria para guardar un valor de punto flotante, pero lo guarda con aproximadamente el doble de precisión de un `float`.
- El método `Double.parseDouble` (un método `static` de la clase `Double`) convierte su argumento `String` en un valor `double` de punto flotante. La clase `Double` forma parte del paquete `java.lang`.
- El método `drawLine` de la clase `Graphics` dibuja líneas. El método requiere cuatro argumentos que representan los dos extremos de la línea en el applet (las coordenadas *x* y *y* del primer extremo en la línea, y las coordenadas *x* y *y* del segundo extremo en la línea). Todos los valores de las coordenadas se especifican con respecto a la coordenada de la esquina superior izquierda *(0, 0)* del applet.

http://librosysolucionarios.net

- El método `drawRect` dibuja un rectángulo con base en sus cuatro argumentos. Los primeros dos valores enteros representan la coordenada superior izquierda *x* y *y*, en donde el objeto `Graphics` empieza a dibujar el rectángulo. Los argumentos tercero y cuarto son enteros no negativos que representan la anchura y la altura del rectángulo en píxeles, respectivamente.
- Para utilizar las características de Java 2 en un applet, Sun proporciona el Complemento Java (Java Plug-in) para pasar por alto el soporte para Java de un navegador y utilizar una versión completa del Entorno Java 2 en tiempo de ejecución (J2RE) que está instalado en el equipo local del usuario.

TERMINOLOGÍA

applet (subprograma)
`applet`, elemento de HTML
`<applet>`, etiqueta
`appletviewer`
archivo de texto
argumento de línea de comandos
`boolean`, tipo primitivo
`byte`, tipo primitivo
campo (variable de instancia)
clase derivada
código fuente
Complemento Java (Java Plug-in)
contenedor de applets
coordenada
crear un objeto
`char`, tipo primitivo
`Double`, clase
`double`, tipo primitivo
`drawLine`, método de `Graphics`
`drawRect`, método de `Graphics`
`drawString`, método de `Graphics`
elemento del HTML
Entorno Java 2 en tiempo de ejecución (JRE)
`extends`, palabra clave
`float`, tipo primitivo
`Graphics`, clase
`height`, altura de un applet
`init`, método de `JApplet`
instanciar un objeto
`int`, tipo primitivo
invocar a un método
`JApplet`, clase
`java.awt`, paquete
`javax.swing`, paquete
Lenguaje de Marcado de Hipertexto (HTML)
lista de parámetros
`long`, tipo primitivo
llamada a un método
mensaje
Microsoft Internet Explorer
navegador
Netscape
número de punto flotante
objeto
`paint`, método de `JApplet`
`parseDouble`, método de `Double`
píxel (elemento de imagen)
referencia
Salir (Quit), elemento de menú en `appletviewer`
`short`, tipo primitivo
`start`, método de `JApplet`
subclase
Subprograma, menú del `appletviewer`
superclase
tipo de referencia
tipo primitivo
variable
variable local
Volver a cargar (Reload), menú en `appletviewer`
`width`, anchura de un applet

EJERCICIOS DE AUTOEVALUACIÓN

3.1 Complete las siguientes oraciones:
a) La clase _____ cuenta con métodos para dibujar.
b) Los applets, de Java empiezan a ejecutarse con una serie de tres llamadas a los métodos: _____, _____ y _____.
c) Los métodos _____ y _____ muestran líneas y rectángulos, respectivamente.
d) La palabra clave _____ indica que una nueva clase es subclase de una clase existente.
e) Todo applet de Java 2 debe extender la clase _____.
f) Los ocho tipos primitivos de Java son _____, _____, _____, _____, _____, _____, _____ y _____.

3.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Para dibujar un rectángulo, el método `drawRect` requiere cuatro argumentos que especifican dos puntos en el applet.
b) El método `drawLine` requiere cuatro argumentos que especifiquen dos puntos en el applet para dibujar una línea.
c) El tipo `Double` es un tipo primitivo.
d) El tipo `int` se utiliza para declarar un número de punto flotante.
e) El método `Double.parseDouble` convierte un valor `String` en un valor primitivo `double`.

3.3 Escriba instrucciones en Java para realizar cada una de las siguientes tareas:
a) Mostrar un cuadro de diálogo que pida al usuario que escriba un número de punto flotante.
b) Convertir una cadena en número de punto flotante y guardar el valor convertido en la variable `edad` de tipo `double`. Suponga que la cadena se guarda en `cadenaValor`.

c) Dibujar el mensaje `"Éste es un programa en Java"` en una línea en la posición *(10, 10)* de un applet (suponga que va a definir esta instrucción en el método `paint` del applet).
d) Dibujar el mensaje `"Éste es un programa en Java"` en dos líneas, empezando en la posición *(10, 10)* en un applet (suponga que estas instrucciones se van a definir en el método `paint` del applet). Haga que la primera línea termine con `Java`. Haga que las dos líneas comiencen en la misma coordenada *x*.

3.4 ¿Cuál es la diferencia entre una variable local y un campo?

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

3.1 a) `Graphics`. b) `init`, `start`, `paint`. c) `drawLine`, `drawRect`. d) `extends`. e) `JApplet`. f) `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`.

3.2 a) Falso. El método `drawRect` requiere cuatro argumentos: dos que especifican la esquina superior izquierda del rectángulo, y dos que especifican la anchura y altura del rectángulo. b) Verdadero. c) Falso. El tipo `Double` es una clase en el paquete `java.lang`; `double` es un tipo de datos primitivo. Recuerde que los nombres que empiezan con letra mayúscula son generalmente nombres de clases. d) Falso. Los tipos `double` o `float` pueden utilizarse para declarar un número de punto flotante. El tipo `int` se utiliza para declarar enteros. e) Verdadero.

3.3 a)
```
cadenaValor = JOptionPane.showInputDialog(
   "Escriba un número de punto flotante" );
```
b)
```
edad = Double.parseDouble( cadenaValor );
```
c)
```
g.drawString( "Éste es un programa en Java", 10, 10 );
```
d)
```
g.drawString( "Éste es un programa en", 10, 10);
g.drawString( "Java", 10, 25 );
```

3.4 Una variable local se declara en el cuerpo de un método, y puede utilizarse solamente desde el punto en el que se declara, hasta el final de la declaración del método. Un campo se declara en una clase, pero no en el cuerpo de sus métodos. Todo objeto (instancia) de una clase tiene una copia separada de los campos de esa clase. Además, los campos son accesibles para todos los métodos de la clase. (Veremos una excepción a esto en el capítulo 8.)

EJERCICIOS

3.5 Complete las siguientes oraciones:
a) El tipo _____ declara una variable de punto flotante con precisión simple.
b) Si la clase `Double` cuenta con el método `parseDouble` para convertir una cadena en `double`, y la clase `Integer` cuenta con el método `parseInt` para convertir una cadena en `int`, entonces probablemente la clase `Float` cuente con el método _____ para convertir una cadena en `float`.
c) El tipo _____ se utiliza para declarar variables de punto flotante con precisión doble.
d) El _____ o un navegador pueden usarse para ejecutar un applet de Java.
e) Para cargar un applet en un navegador, primero se debe definir un archivo _____.
f) Las etiquetas _____ y _____ de HTML especifican que un applet debe cargarse y ejecutarse en un contenedor de applets.

3.6 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) La aplicación que ejecuta un applet generalmente se conoce como un contenedor de applets.
b) Al utilizar una declaración `import` de la forma `javax.swing.*`, se importan todas las clases en el paquete.
c) Usted no necesita declaraciones `import` si se especifica el nombre completo del paquete y de la clase cada vez que haga referencia a esa clase en un programa.

3.7 Escriba un applet que pida al usuario que escriba dos números de punto flotante, obtenga los dos números del usuario y dibuje la suma, producto (multiplicación), diferencia y cociente (división) de los dos números. Utilice las técnicas que se muestran en la figura 3.13.

3.8 Escriba un applet que pida al usuario que escriba dos números de punto flotante, obtenga los números del usuario y muestre el número mayor, seguido por las palabras `"es mayor"`, como una cadena en el applet. Si los números son iguales, debe imprimir el mensaje `"Éstos números son iguales"`. Utilice las técnicas que se muestran en la figura 3.13.

3.9 Escriba un applet que reciba tres números de punto flotante del usuario y que muestre la suma, promedio, producto, menor y mayor de estos números como cadenas en el applet. Utilice las técnicas que se muestran en la figura 3.13.

3.10 Escriba un applet que pida al usuario que escriba el radio de un círculo como número de punto flotante y que dibuje el diámetro, circunferencia y área del círculo. Use el valor 3.14159 para π. Utilice las técnicas que se muestran en

http://librosysolucionarios.net

la figura 3.13. [*Nota:* También puede usar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` está definida en el paquete `java.lang`, por lo que no necesita importarla.] Use las siguientes fórmulas (r es el radio):

diámetro = $2r$
circunferencia = $2\pi r$
área = πr^2

3.11 Escriba un applet que lea cinco enteros, determine e imprima los enteros mayor y menor en el grupo. Use solamente las técnicas de programación que aprendió en este capítulo y en el capítulo 2. Dibuje los resultados en el applet.

3.12 ¿Qué es lo que imprime el siguiente código?

```
g.drawString( "*", 25, 25);
g.drawString( "***", 25, 55);
g.drawString( "*****", 25, 85);
g.drawString( "***", 25, 70);
g.drawString( "*", 25, 40);
```

3.13 Escriba un applet que dibuje un patrón de tablero de damas, como se muestra a continuación:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

3.14 Escriba un applet que dibuje rectángulos con distintos tamaños y ubicaciones.

3.15 Escriba un applet que permita al usuario introducir valores para los argumentos requeridos por el método `drawRect`, y que después dibuje un rectángulo utilizando los cuatro valores introducidos.

3.16 La clase `Graphics` contiene el método `drawOval`, el cual toma los mismos cuatro argumentos que el método `drawRect`. Los argumentos para el método `drawOval` especifican el "cuadro delimitador" para el óvalo: los lados del cuadro delimitador son los límites del óvalo. Escriba un applet en Java que dibuje el óvalo y un rectángulo con los mismos cuatro argumentos. El óvalo debe tocar el rectángulo en el centro de cada lado.

3.17 Modifique la solución al ejercicio 3.16 para producir óvalos de distintas formas y tamaños.

3.18 Escriba un applet que permita al usuario introducir los cuatro argumentos requeridos por el método `drawOval`, y que después dibuje un óvalo utilizando los cuatro valores introducidos.

3.19 Utilizando solamente las técnicas de programación de este capítulo y del capítulo 2, escriba un applet que calcule los cuadrados y cubos de los números del 0 al 10, y que dibuje los valores resultantes en el formato de tabla que se muestra a continuación. [*Nota:* Este programa no requiere de ninguna entrada por parte del usuario.]

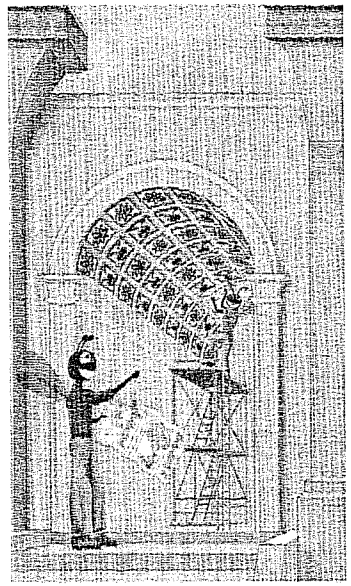
```
número      cuadrado    cubo
0           0           0
1           1           1
2           4           8
3           9           27
4           16          64
5           25          125
6           36          216
7           49          343
8           64          512
9           81          729
10          100         1000
```

4

Instrucciones de control: Parte 1

Objetivos

- Comprender las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos mediante el proceso de mejoramiento de arriba a abajo, paso a paso.
- Utilizar las estructuras de selección `if` e `if...else` para elegir entre distintas acciones alternativas.
- Utilizar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Comprender la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de asignación, incremento y decremento.



Desplacémonos un lugar.
Lewis Carroll

La rueda se convirtió en un círculo completo.
William Shakespeare

¡Cuántas manzanas tuvieron que caer en la cabeza de Newton antes de que entendiera el suceso!
Robert Frost

http://librosysolucionarios.net

Plan general

4.1 Introducción
4.2 Algoritmos
4.3 Seudocódigo
4.4 Estructuras de control
4.5 Estructura de selección `if` simple
4.6 Estructura de selección `if...else`
4.7 Estructura de repetición `while`
4.8 Cómo formular algoritmos: Ejemplo práctico 1 (repetición controlada por un contador)
4.9 Cómo formular algoritmos mediante el mejoramiento de arriba a abajo, paso a paso: Ejemplo práctico 2 (repetición controlada por un centinela)
4.10 Cómo formular algoritmos mediante el mejoramiento de arriba a abajo, paso a paso: Ejemplo práctico 3 (estructuras de control anidadas)
4.11 Operadores de asignación compuestos
4.12 Operadores de incremento y decremento
4.13 Tipos primitivos
4.14 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar los atributos de las clases

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, es imprescindible tener una comprensión detallada de todo el problema, además de una metodología cuidadosamente planeada para resolver el problema. Al escribir un programa, es igualmente esencial comprender los tipos de bloques de construcción disponibles, y emplear los principios comprobados para construir programas. En este capítulo y en el 5 hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la programación estructurada. Las técnicas que aprenda aquí son aplicables a Java y a la mayoría de los otros lenguajes de alto nivel. Cuando en el capítulo 8 estudiemos de manera más detallada la programación basada en objetos veremos que los conceptos aquí presentados son útiles para crear clases y manipular objetos.

4.2 Algoritmos

Cualquier problema de computación puede resolverse ejecutando una serie de acciones en un orden específico. El *procedimiento* para resolver un problema en términos de

1. las *acciones* a ejecutar y
2. el *orden* en el que se ejecutan estas acciones

se conoce como un *algoritmo*. El siguiente ejemplo demuestra que es importante especificar de manera correcta el orden en el que se ejecutan las acciones.

Considere el algoritmo "levantarse y arreglarse" que sigue un ejecutivo para levantarse de la cama e ir a trabajar: (1) levantarse; (2) quitarse la pijama; (3) bañarse; (4) vestirse; (5) desayunar; (6) transportarse al trabajo. Esta rutina logra que el ejecutivo llegue al trabajo bien preparado para tomar decisiones críticas. Suponga que los mismos pasos se realizan en un orden ligeramente distinto: (1) levantarse; (2) quitarse la pijama; (3) vestirse; (4) bañarse; (5) desayunar; (6) transportarse al trabajo. En este caso nuestro ejecutivo llegará al trabajo todo mojado.

http://librosysolucionarios.net

Al proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa, se le llama *control del programa*.

4.3 Seudocódigo

El *seudocódigo* es un lenguaje informal que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java. El seudocódigo que presentaremos en este libro es especialmente útil para desarrollar algoritmos que se convertirán en porciones estructuradas de programas en Java. El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable con el usuario, aunque no es realmente un lenguaje de programación de computadoras.

El seudocódigo no se ejecuta en las computadoras, pero ayuda al programador a "organizar" un programa antes de intentar escribirlo en un lenguaje de programación como Java. Este capítulo presenta varios ejemplos de cómo utilizar el seudocódigo efectivamente en el desarrollo de programas en Java.

El estilo de seudocódigo que presentaremos consiste sólo en caracteres, de manera que los programadores pueden escribir el seudocódigo de manera conveniente, utilizando cualquier programa editor. Un programa en seudocódigo preparado de manera cuidadosa puede convertirse fácilmente en su correspondiente programa en Java. En muchos casos, esto requiere tan sólo reemplazar las instrucciones en seudocódigo con sus instrucciones equivalentes en Java.

Por lo general, el seudocódigo describe sólo las *instrucciones ejecutables*; las acciones que ocurren después de que el programador convierte un programa de seudocódigo a Java y el programa se ejecuta en la computadora. Las declaraciones no son instrucciones ejecutables. Por ejemplo, la declaración

```
int valorEntero;
```

indica al compilador el tipo de la variable `valorEntero`, e instruye a Java para que reserve espacio en memoria para esta variable. Cuando el programa se ejecuta, esta declaración no provoca ninguna acción, como entrada, salida o un cálculo. Por lo tanto, esta declaración no necesita aparecer en el seudocódigo. Sin embargo, algunos programadores optan por enlistar variables y mencionar sus propósitos al principio de su seudocódigo.

4.4 Estructuras de control[1]

Por lo general, en un programa las instrucciones se ejecutan una después de otra en el orden en que están escritas. Este proceso se conoce como *ejecución secuencial*. Varias instrucciones en Java permiten al programador especificar que la siguiente instrucción a ejecutarse tal vez no sea la siguiente en la secuencia. Esto se conoce como *transferencia de control*.

Durante la década de los sesenta, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la *instrucción* `goto` (utilizada en varios lenguajes de programación, incluyendo C y Basic), la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa. La noción de la llamada *programación estructurada* se hizo casi un sinónimo de la "eliminación del `goto`".[2]

Las investigaciones de Bohm y Jacopini[3] demostraron que los programas podían escribirse sin instrucciones `goto`. El reto de la época para los programadores fue cambiar sus estilos a una "programación sin `goto`". No fue sino hasta la década de los setenta cuando los programadores tomaron en serio la programación estructurada. Los resultados fueron impresionantes, a medida que los grupos de desarrollo de software reportaron reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros fue que los programas estructurados

1. El término "estructuras de control" viene del campo de la ciencia computacional. Cuando presentemos las implementaciones de Java para las estructuras de control, nos referiremos a ellas con la terminología de la Especificación del lenguaje Java, que se refiere a ellas como "instrucciones".
2. Java no tiene una instrucción `goto`; sin embargo, el nombre `goto` está reservado por Java y no debe usarse como identificador en los programas.
3. Bohm, C. y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, Mayo de 1996, páginas 336-371.

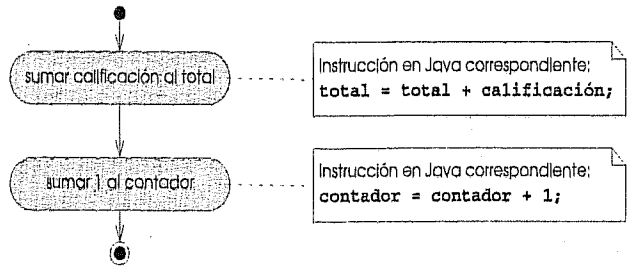


Figura 4.1 Diagrama de actividad de una estructura de secuencia.

eran más claros, más fáciles de depurar y modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

El trabajo de Bohm y Jacopini demostró que todos los programas podían escribirse en términos de tres estructuras de control solamente: la *estructura de secuencia*, la *estructura de selección* y la *estructura de repetición*. La estructura de secuencia está integrada en Java. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones en Java una después de otra, en el orden en que estén escritas. El *diagrama de actividad* de la figura 4.1 ilustra una estructura de secuencia típica, en la que se realizan dos cálculos en orden. Java permite tantas acciones como deseemos en una estructura de secuencia. Como veremos más adelante, en donde quiera que se coloque una sola acción, podrán colocarse varias acciones en secuencia.

Los diagramas de actividad son parte del *Lenguaje Unificado de Modelado (UML, por sus siglas en inglés)*, un estándar industrial para modelar sistemas de software.[4] Un diagrama de actividad modela el *flujo de trabajo* (también conocido como la *actividad*) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 4.1. Los diagramas de actividad están compuestos por símbolos de propósito especial, como los *símbolos de estado de acción* (un rectángulo en el que sus lados izquierdo y derecho se reemplazan con arcos hacia fuera), *rombos (diamantes)* y *pequeños círculos*; estos símbolos se conectan mediante *flechas de transición*, que representan el flujo de la actividad.

Al igual que el seudocódigo, los diagramas de actividad ayudan a los programadores a desarrollar y representar algoritmos; sin embargo, muchos de ellos aún prefieren el seudocódigo. Los diagramas de actividad muestran claramente cómo operan las estructuras de control.

Considere el diagrama de actividad para la estructura de secuencia de la figura 4.1. Este diagrama contiene dos *estados de acción* que representan las acciones a realizar. Cada estado de acción contiene una *expresión de acción* (por ejemplo, "sumar calificación a total" o "sumar 1 al contador") que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad se llaman flechas de transición o de índice. Estas flechas representan *transiciones*, las cuales indican el orden en el que ocurren las acciones representadas por los estados de acción; el programa que implementa las actividades ilustradas por el diagrama de actividad de la figura 4.1 primero suma `calificación` a `total`, y después suma `1` a `contador`.

El *círculo relleno* que se encuentra en la parte superior del diagrama de actividad representa el *estado inicial* de la actividad, es decir, el inicio del flujo de trabajo antes de que el programa realice las actividades modeladas. El círculo sólido (también llamado diana) rodeado por una circunferencia que aparece en la parte inferior del diagrama de actividad representa el *estado final*, es decir, el final del flujo de trabajo después de que el programa realiza sus actividades.

La figura 4.1 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a éstos rectángulos se les llama *notas*. Las notas son comentarios con explicaciones que describen el propósito de

4. Para obtener más información sobre UML vea nuestro ejemplo práctico opcional, el cual aparece al final de los capítulos 8, 10 a 14, 16 y 19, o visite `www.uml.org`.

los símbolos en el diagrama. La figura 4.1 utiliza las notas de UML para mostrar el código en Java asociado con cada uno de los estados de acción en el diagrama de actividad. Una *línea punteada* conecta cada nota con el elemento que ésta describe. Los diagramas de actividad generalmente no muestran el código en Java que implementa la actividad. En este libro utilizamos las notas con este propósito, para mostrar cómo se relaciona el diagrama con el código en Java.

Java tiene tres tipos de estructuras de selección (las cuales se describen en este capítulo y en el siguiente). La instrucción `if` realiza (selecciona) una acción si la condición es verdadera, o evita la acción si la condición es falsa. La instrucción `if...else` realiza una acción si la condición es verdadera, o realiza una acción distinta si la condición es falsa. La instrucción `switch` (capítulo 5) realiza una de entre varias acciones, dependiendo del valor de una expresión.

La instrucción `if` es una *estructura de selección simple*, ya que selecciona o ignora una sola acción (o, como pronto veremos, un solo grupo de acciones). La instrucción `if...else` es una *estructura de selección doble*, ya que selecciona entre dos acciones distintas (o grupos de acciones). La instrucción `switch` es una *estructura de selección múltiple*, ya que selecciona entre diversas acciones (o grupos de acciones).

Java cuenta con tres estructuras de repetición (también llamadas *estructuras de ciclo*) que permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición (llamada la *condición de continuación del ciclo*) siga siendo verdadera. Las estructuras de repetición se implementan con las instrucciones `while`, `do...while` y `for`.[5] Las instrucciones `while` y `for` realizan la acción (o grupo de acciones) en sus cuerpos, cero o más veces; si la condición de continuación del ciclo es inicialmente falsa, no se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) en su cuerpo, una o más veces.

Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son *palabras clave* en Java. Estas palabras se utilizan para implementar varias características de Java, como las estructuras de control. Las palabras clave no pueden usarse como identificadores, por ejemplo, como nombres de variables. En la figura 4.2 se muestra una lista completa de las palabras clave en Java. Además de las palabras clave, Java también contiene las *palabras reservadas* `true`, `false` y `null`, las cuales representan valores especiales. Éstas tampoco pueden usarse como identificadores.



Error común de programación 4.1

Utilizar una palabra clave como identificador es un error de sintaxis.

Palabras clave de Java

`abstract`	`assert`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`	`continue`
`default`	`do`	`double`	`else`	`extends`
`final`	`finally`	`float`	`for`	`if`
`implements`	`import`	`instanceof`	`int`	`interface`
`long`	`native`	`new`	`package`	`private`
`protected`	`public`	`return`	`short`	`static`
`strictfp`	`super`	`switch`	`synchronized`	`this`
`throw`	`throws`	`transient`	`try`	`void`
`volatile`	`while`			

Palabras que están reservadas, pero que no se utilizan actualmente

`const`	`goto`

Figura 4.2 Palabras clave de Java.

5. El capítulo 5 presenta las estructuras de repetición `do...while` y `for`.

http://librosysolucionarios.net



Error común de programación 4.2

Colocar un carácter de espacio en una palabra clave (por ejemplo, `instance of`) es un error de sintaxis.

Entonces, Java tiene sólo tres estructuras de control: la secuencia de estructura, la estructura de selección (tres tipos) y la estructura de repetición (tres tipos). Cada programa se forma combinando tantas estructuras de secuencia, selección y repetición como sea apropiado para el algoritmo que implemente el programa. Al igual que con la estructura de secuencia de la figura 4.1, podemos modelar cada una de las estructuras de control como un diagrama de actividad. Cada diagrama contiene un estado inicial y final, los cuales representan el punto de entrada y salida de una estructura de control, respectivamente. Las *estructuras de una sola entrada/una sola salida (o controles de entrada sencilla/salida sencilla)* facilitan la creación de programas; las estructuras de control están unidas entre sí mediante la conexión del punto de salida de una estructura de control, al punto de entrada de la siguiente. Este procedimiento es similar a la manera en que un niño apila los bloques de construcción, así que a esto le llamamos *apilamiento de estructuras de control*. En breve aprenderemos que sólo hay una manera alternativa de conectar las estructuras de control: *anidamiento de estructuras de control*, en el cual una estructura de control aparece dentro de otra estructura de control. Por lo tanto, los algoritmos en los programas en Java se crean a partir de sólo tres tipos principales de estructuras de control, que se combinan sólo de dos formas.

4.5 Estructura de selección `if` simple

Los programas utilizan las estructuras de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es de 60. La instrucción en seudocódigo

Si la calificación del estudiante es mayor o igual a 60
Imprimir "Aprobado"

determina si la condición "la calificación del estudiante es mayor o igual a 60" es verdadera o falsa. Si la condición es verdadera se imprime "Aprobado", y se "ejecuta" en orden la siguiente instrucción en seudocódigo. (Recuerde que el seudocódigo no es un verdadero lenguaje de programación.) Si la condición es falsa se ignora la instrucción Imprimir, y se ejecuta en orden la siguiente instrucción en seudocódigo. La sangría de la segunda línea de esta estructura de selección es opcional, pero se recomienda ya que enfatiza la estructura inherente de los programas estructurados.

La instrucción anterior *if* en seudocódigo puede escribirse en Java de la siguiente manera:

```
if ( calificacionEstudiante >= 60 )
   System.out.println( "Aprobado" );
```

Observe que el código en Java corresponde en gran medida con el seudocódigo. Ésta es una de las propiedades que hace del seudocódigo una herramienta de programación tan útil.

La figura 4.3 muestra la instrucción `if` de selección simple. Este diagrama de actividad contiene lo que quizá sea el símbolo más importante en un diagrama de este tipo, el rombo o *símbolo de decisión*, el cual indica que se va a tomar una decisión. Un símbolo de decisión indica que el flujo de trabajo continuará a lo largo de una ruta determinada por las *condiciones de guardia* (*condiciones de seguridad* o *símbolos de guardia*, como también se les conoce) asociadas de ese símbolo, que pueden ser verdaderas o falsas.[6] Cada flecha de transición que sale de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, encima o a un lado de la flecha de transición). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición. En la figura 4.3, si la calificación es mayor o igual a 60, el programa imprime "Aprobado" y luego se dirige al estado final de esta actividad. Si la calificación es menor a 60, el programa se dirige inmediatamente al estado final sin mostrar ningún mensaje.

La instrucción `if` es una estructura de control de una sola entrada/una sola salida. Pronto veremos que los diagramas de actividad para las estructuras de control restantes también contienen estados iniciales, flechas de

6. En el capítulo 2 aprendimos que las decisiones pueden basarse en condiciones que contengan operadores relacionales o de igualdad. En realidad, una decisión puede tomarse en base a cualquier expresión que se evalúe en uno de los valores permitidos en el tipo `boolean` de Java (es decir, cualquier expresión que dé cómo resultado `true` o `false`).

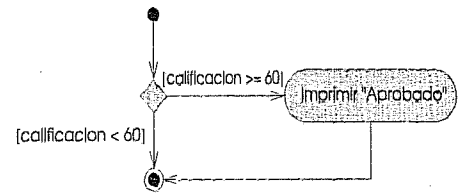


Figura 4.3 Diagrama de actividad de la instrucción `if` de selección simple.

transición, estados de acción que indican las acciones a realizar, símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar y estados finales. Esto es consistente con el *modelo de programación acción/decisión* que hemos estado enfatizando.

Imagine siete cajones, en donde cada uno contiene sólo una de las estructuras de control de Java. Estas instrucciones de control están vacías. Su tarea es ensamblar un programa a partir de tantas estructuras de control de cada tipo como lo requiera el algoritmo, combinando esas estructuras de control en sólo dos formas posibles (apilando o anidando), y después llenando los estados de acción y las decisiones con expresiones de acción y condiciones de guardia, en una manera que sea apropiada para el algoritmo. Hablaremos sobre la variedad de formas en que pueden escribirse las acciones y las decisiones.

4.6 Estructura de selección `if...else`

La instrucción `if` de selección simple realiza una acción indicada solamente cuando la condición es verdadera (`true`); de no ser así, se evita dicha acción. La instrucción `if..else` de selección doble permite al programador especificar una acción a realizar cuando la condición es verdadera, y otra distinta cuando la condición es falsa. Por ejemplo, la instrucción en seudocódigo:

Si la calificación del estudiante es mayor o igual a 60
Imprimir "Aprobado"
De lo contrario
Imprimir "Reprobado"

imprime "Aprobado" si la calificación del estudiante es mayor o igual a 60 e imprime "Reprobado" si la calificación del estudiante es menor a 60. En cualquier caso, después de que ocurre la impresión se "ejecuta" la siguiente instrucción en seudocódigo en la secuencia.

La instrucción anterior *if...else* en seudocódigo puede escribirse en Java como

```
if ( calificación >= 60 )
   System.out.println( "Aprobado" );
else
   System.out.println( "Reprobado" );
```

Observe que el cuerpo de la instrucción `else` también tiene sangría. Cualquiera que sea la convención de sangría que usted elija, debe aplicarla consistentemente en todos sus programas. Es difícil leer programas que no obedecen las convenciones de espaciado uniformes.



Buena práctica de programación 4.1

Utilice sangría en ambos cuerpos de instrucciones de una estructura `if...else`.



Buena práctica de programación 4.2

Si hay varios niveles de sangría, en cada nivel debe aplicarse la misma cantidad de espacio adicional.

La figura 4.4 muestra el flujo de control en la instrucción `if...else`. Una vez más (además del estado inicial, las flechas de transición y el estado final), los símbolos en el diagrama de actividad representan estados de acción y decisiones. Nosotros seguimos enfatizando este modelo de computación acción/decisión. Imagine de

http://librosysolucionarios.net

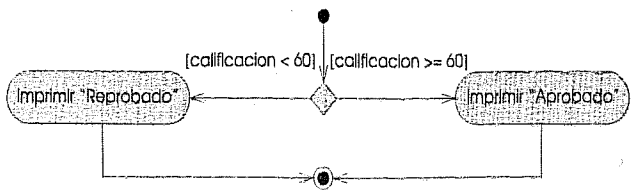


Figura 4.4 Diagrama de actividad de la instrucción `if...else` de selección doble.

nuevo un cajón profundo que contiene tantas instrucciones `if...else` vacías como sea necesario para crear cualquier programa en Java. Su trabajo es ensamblar estas instrucciones `if...else` (apilando o anidando) con cualquier otra estructura de control requerida por el algoritmo. Usted debe llenar los estados de acción y los símbolos de decisión con expresiones de acción y condiciones de guardia que sean apropiadas para el algoritmo.

Operador condicional (?:)

Java cuenta con el *operador condicional* (`?:`), que en ocasiones puede utilizarse en lugar de una instrucción `if...else`. Éste es el único *operador ternario* en Java; es decir, que utiliza tres operandos. En conjunto, los operandos y `? :` forman una *expresión condicional*. El primer operando (a la izquierda del `?`) es una expresión `boolean` (es decir, una condición), el segundo operando (entre el `?` y `:`) es el valor de la expresión condicional si la expresión `boolean` es verdadera, y el tercer operando (a la derecha del `:`) es el valor de la expresión condicional si la expresión `boolean` se evalúa como `false`. Por ejemplo, la instrucción

```
System.out.println( calificacionEstudiante >= 60 ? "Aprobado" : "Reprobado" );
```

imprime el valor del argumento de `println`, que es una expresión condicional. La expresión condicional en esta instrucción produce como resultado la cadena `"Aprobado"` si la expresión `boolean calificacionEstudiante >= 60` es verdadera, o produce como resultado la cadena `"Reprobado"` si la expresión `boolean` es falsa. Por lo tanto, la instrucción con el operador condicional realiza en esencia la misma función que la instrucción `if...else` que se mostró anteriormente, en esta sección. La precedencia del operador condicional es baja, por lo que toda la expresión condicional se coloca normalmente entre paréntesis. Pronto veremos que las expresiones condicionales pueden usarse en algunas situaciones en las que no se puede utilizar instrucciones `if...else`.



Buena práctica de programación 4.3

En general, las expresiones condicionales son más difíciles de leer que las instrucciones `if...else`. En la mayoría de los casos, las expresiones condicionales deben usarse para reemplazar sólo a las instrucciones `if...else` simples que seleccionan uno de dos valores.

Instrucciones `if...else` anidadas

Un programa puede evaluar varios casos colocando instrucciones `if...else` dentro de otra instrucción `if...else` para crear *instrucciones `if...else` anidadas*. Por ejemplo, el siguiente seudocódigo representa una instrucción `if...else` anidada que imprime `A` para las calificaciones de exámenes mayores o iguales a 90, `B` para las calificaciones en el rango de 80 a 89, `C` para las calificaciones en el rango de 70 a 79, `D` para las calificaciones en el rango de 60 a 69 y `F` para todas las demás calificaciones:

```
Si la calificación del estudiante es mayor o igual a 90
    Imprimir "A"
De lo contrario
    Si la calificación del estudiante es mayor o igual a 80
        Imprimir "B"
    De lo contrario
        Si la calificación del estudiante es mayor o igual a 70
            Imprimir "C"
        De lo contrario
            Si la calificación del estudiante es mayor o igual a 60
                Imprimir "D"
            De lo contrario
                Imprimir "F"
```


Este seudocódigo puede escribirse en Java como

```
if ( calificacionEstudiante >= 90 )
   System.out.println( "A" );
else
   if ( calificacionEstudiante >= 80 )
      System.out.println( "B" );
   else
      if ( calificacionEstudiante >= 70 )
         System.out.println( "C" );
      else
         if ( calificacionEstudiante >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

Si `calificacionEstudiante` es mayor o igual a 90, las primeras cuatro condiciones serán verdaderas, pero sólo se ejecutará la instrucción en la parte `if` de la primera instrucción `if...else`. Después de que se ejecute esa instrucción, se evita la parte `else` de la instrucción `if...else` más "externa". La mayoría de los programadores en Java prefieren escribir la instrucción `if...else` anterior así:

```
if ( calificacionEstudiante >= 90 )
   System.out.println( "A" );
else if ( calificacionEstudiante >= 80 )
   System.out.println( "B" );
else if ( calificacionEstudiante >= 70 )
   System.out.println( "C" );
else if ( calificacionEstudiante >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular ya que evita usar mucha sangría hacia la derecha en el código. Dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se dividan y empeorando la legibilidad del programa.

Problema del `else` suelto

El compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves (`{}`). Este comportamiento puede ocasionar lo que se conoce como el *problema del `else` suelto*. Por ejemplo,

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x e y son > 5" );
else
   System.out.println( "x es <= 5" );
```

Parece indicar que si `x` es mayor que `5`, la instrucción `if` anidada determina si `y` es también mayor que `5`. De ser así, se produce como resultado la cadena `"x e y son > 5"`. De lo contrario, *parece ser* que si `x` no es mayor que `5`, la instrucción `else` que es parte del `if...else` produce como resultado la cadena `"x es <= 5"`.

¡Cuidado! Esta instrucción `if...else` anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x e y son > 5" );
   else
      System.out.println( "x es <= 5" );
```

http://librosysolucionarios.net

en donde el cuerpo del primer `if` es un `if...else` anidado. Esta instrucción evalúa si `x` es mayor que `5`. De ser así, la ejecución continúa evaluando si `y` es también mayor que `5`. Si la segunda condición es verdadera, se muestra la cadena apropiada (`"x e y son > 5"`). No obstante, si la segunda condición es falsa se muestra la cadena `"x es <= 5"`, aún y cuando sabemos que `x` es mayor que `5`.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, debe escribirse de la siguiente manera:

```
if ( x > 5 ) {
   if ( y > 5 )
      System.out.println( "x e y son > 5" );
}
else
   System.out.println( "x es <= 5" );
```

Las llaves ({}) indican al compilador que la segunda instrucción `if` se encuentra en el cuerpo del primer `if`, y que el `else` está asociado con el primer `if`. Los ejercicios 4.21 y 4.22 investigan con más detalle el problema del `else` suelto.

Bloques

La instrucción `if` normalmente espera sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves ({ y }). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama *bloque*.



Observación de Ingeniería de software 4.1

Un bloque puede colocarse en cualquier parte de un programa en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if ( calificacion >= 60 )
   System.out.println( "Aprobado" );
else {
   System.out.println( "Reprobado." );
   System.out.println( "Debe tomar este curso otra vez." );
}
```

En este caso, si `calificacion` es menor que `60`, el programa ejecuta ambas instrucciones en el cuerpo del `else` e imprime

```
Reprobado.
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
System.out.println ( "Debe tomar este curso otra vez." );
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if...else` y se ejecutaría sin importar que la calificación fuera menor a 60.

Los errores de sintaxis (como cuando se omite una llave en un bloque del programa) los atrapa el compilador. Un *error lógico* (como cuando se omiten ambas llaves en un bloque del programa) tiene su efecto en tiempo de ejecución. Un *error lógico fatal* hace que un programa falle y termine antes de tiempo. Un *error lógico no fatal* permite que un programa siga ejecutándose, pero éste produce resultados incorrectos.



Error común de programación 4.3

Olvidar una o las dos llaves que delimitan un bloque puede provocar errores de sintaxis o errores lógicos en un programa.



Buena práctica de programación 4.4

Colocar siempre las llaves en una instrucción `if...else` (o cualquier estructura de control) ayuda a evitar que se omitan de manera accidental, en especial, cuando posteriormente se agregan instrucciones a una cláusula `if` o `else`. Para evitar que esto suceda, algunos programadores prefieren escribir la llave inicial y la final de los bloques antes de escribir las instrucciones individuales dentro de ellas.

Así como un bloque puede colocarse en cualquier parte en donde pueda colocarse una sola instrucción individual, también es posible no tener instrucción alguna; a esto se le llama una *instrucción vacía* (o *instrucción nula*). La instrucción vacía se representa colocando un punto y coma (`;`) en donde normalmente iría una instrucción.



Error común de programación 4.4

Colocar un punto y coma después de la condición en una instrucción `if...else` produce un error lógico en las instrucciones `if` de selección simple, y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

4.7 Estructura de repetición `while`

Una *estructura de repetición* (también llamada *estructura de ciclo*) permite al programador especificar que un programa debe repetir una acción mientras cierta condición sea verdadera. La instrucción en seudocódigo

> ***Mientras existan más artículos en mi lista de compras***
> ***Comprar el siguiente artículo y quitarlo de mi lista***

describe la repetición que ocurre durante una salida de compras. La condición "existan más artículos en mi lista de compras" puede ser verdadera o falsa. Si es verdadera, entonces se realiza la acción "Comprar el siguiente artículo y quitarlo de mi lista". Esta acción se realizará en forma repetida mientras la condición sea verdadera. La instrucción (o instrucciones) contenida en la estructura de repetición *while* constituye el cuerpo de esta estructura, el cual puede ser una sola instrucción o un bloque. En algún momento, la condición será falsa (cuando el último artículo de la lista de compras sea adquirido y eliminado de la lista). En este punto la repetición terminará y se ejecutará la primera instrucción que esté después de la estructura de repetición.

Como ejemplo de la instrucción `while` en Java, considere un segmento de programa diseñado para encontrar la primera potencia de 2 que sea mayor a 1000. Suponga que la variable `producto` de tipo `int` se inicializa en `2`. Cuando la siguiente instrucción `while` termine de ejecutarse, `producto` contendrá el resultado:

```
int producto = 2;

while ( producto <= 1000 )
   producto = 2 * producto;
```

Cuando esta instrucción `while` comienza a ejecutarse, el valor de la variable `producto` es 2. Cada iteración de la instrucción `while` multiplica a `producto` por 2, por lo que `producto` toma los valores de 4, 8, 16, 32, 64, 128, 256, 512 y 1024, sucesivamente. Cuando la variable `producto` se vuelve 1024, la condición de la instrucción `while` (`producto <= 1000`) se torna falsa. Esto termina la repetición, por lo que el valor final de `producto` es 1024. En este punto, la ejecución del programa continuaría con la siguiente instrucción después de la instrucción `while`.



Error común de programación 4.5

Si no se proporciona, en el cuerpo de una instrucción `while`, una acción que ocasione que en algún momento la condición de un `while` se torne falsa, por lo general se producirá un error lógico conocido como ciclo infinito, en el que el ciclo nunca terminará.

El diagrama de actividad de la figura 4.5 muestra el flujo de control que corresponde a la instrucción `while` anterior. Una vez más (aparte del estado inicial, las flechas de transición, un estado final y dos notas), los símbolos en el diagrama representan un estado de acción y una decisión. Este diagrama también introduce el *símbolo de fusión* de UML. UML representa tanto al símbolo de fusión como al símbolo de decisión como rombos. El símbolo de fusión une dos flujos de actividad en un solo flujo de actividad. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose). Aunque UML representa a los símbolos de decisión y de fusión con la figura del rombo, los símbolos pueden diferenciarse por el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más flechas de transición que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Además, cada flecha de transición que apunta hacia fuera de un

http://librosysolucionarios.net

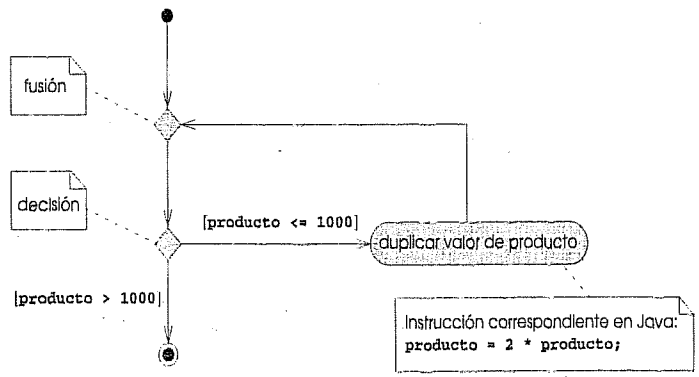


Figura 4.5 Diagrama de actividad de la instrucción de repetición `while`.

símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. Observe que, a diferencia del símbolo de decisión, el símbolo de fusión no tiene su contraparte en código de Java.

Imagine un cajón de instrucciones `while` vacías que pueden apilarse y anidarse con otras estructuras de control para formar una implementación estructurada del flujo de control de un algoritmo. Usted completa los estados de acción y símbolos de decisión con expresiones de acción y condiciones de guardia que sean apropiadas para el algoritmo. El diagrama muestra claramente la repetición de la instrucción `while`. La flecha de transición que emerge del estado de acción apunta a la fusión. De la fusión, el flujo del programa se dirige nuevamente a la decisión que se evalúa al principio de cada iteración en el ciclo. El ciclo continúa ejecutándose hasta que la condición de guardia `producto > 1000` se torna verdadera. Entonces, termina la instrucción `while` (llega a su estado final) y el control pasa a la siguiente instrucción del programa.

4.8 Cómo formular algoritmos: Ejemplo práctico 1 (repetición controlada por un contador)

Para ilustrar la forma en que se desarrollan los algoritmos, resolveremos dos variantes de un problema que promedia las calificaciones de unos estudiantes. Analicemos el siguiente enunciado del problema:

> *A una clase de diez estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para su análisis. Determine el promedio de la clase para este examen.*

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada una de las calificaciones, llevar el registro del total de las calificaciones introducidas, realizar el cálculo para promediar e imprimir el resultado.

Emplearemos seudocódigo para enlistar las acciones a ejecutar y especificar el orden en el que deben ejecutarse. Usaremos una *repetición controlada por contador* para introducir las calificaciones, una por una. Esta técnica utiliza una variable llamada *contador* (o *variable de control*) para controlar el número de veces que debe ejecutarse un conjunto de instrucciones. A la repetición controlada por contador se le llama comúnmente *repetición definida*, ya que el número de repeticiones se conoce antes de que el ciclo empiece a ejecutarse. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta un algoritmo de seudocódigo (figura 4.6) y su correspondiente programa en Java (figura 4.7). La sección 4.9 muestra cómo utilizar el seudocódigo para desarrollar un algoritmo.

Observación de ingeniería de software 4.2



La experiencia ha demostrado que la parte más difícil para la resolución de un problema en una computadora es desarrollar el algoritmo para la solución. Por lo general, una vez que se ha especificado el algoritmo correcto, el proceso de producir un programa funcional en Java a partir de dicho algoritmo es relativamente sencillo.

Observe las referencias en el algoritmo para un total y un contador. La variable *total* se utiliza para acumular la suma de varios valores. La variable *contador* se utiliza para contar; en este caso, para contar el número de calificaciones introducidas. Por lo general, las variables que se utilizan para guardar totales deberían inicializarse en cero antes de utilizarse en un programa.

La línea 3 importa la clase `JOptionPane`, de manera que el programa pueda utilizar los cuadros de diálogo de entrada y de mensaje que se presentaron en el capítulo 2. La línea 5 comienza la declaración de la clase `Promedio1` de la aplicación. Recuerde de la sección 2.2 que la declaración de la clase de una aplicación debe contener un método `main` (líneas 7 a 44), en donde la aplicación comience a ejecutarse.

Las líneas 9 a 14 declaran las variables `total`, `contadorCalif`, `calificación` y `promedio` para que sean de tipo `int`, y la variable `cadenaCalif` de tipo `String`. La variable `cadenaCalif` guarda la cadena que el usuario escribe en el cuadro de diálogo de entrada. La variable `calificación` guarda el valor entero de `cadenaCalif` después de que el programa la convierte en un `int`.

Observe que las declaraciones (en las líneas 9 a 14) aparecen en el cuerpo del método `main`. Recuerde de la sección 3.5 que las variables declaradas en el cuerpo de un método son *variables locales* y pueden utilizarse solamente desde la línea de su declaración en el método, hasta la llave derecha de cierre (`}`) de la declaración del método. La declaración de una variable local debe aparecer antes de que la variable se utilice en ese método. Por ejemplo, las líneas 9 a 14 deben aparecer antes de las líneas 17-40 en la figura 4.7. Una variable local no puede utilizarse fuera del método en el que se declara.

Asignar a total el valor de cero
Asignar al contador de calificaciones el valor de uno

Mientras que el contador de calificaciones sea menor o igual a diez
    Introducir la siguiente calificación
    Sumar la calificación al total
    Sumar uno al contador de calificaciones

Asignar al promedio de la clase el total dividido entre diez
Imprimir el promedio de la clase

Figura 4.6 Algoritmo en seudocódigo que utiliza la repetición controlada por contador para resolver el problema del promedio de una clase.

```
1  // Fig. 4.7: Promedio1.java
2  // Programa del promedio de una clase mediante la repetición controlada por contador.
3  import javax.swing.JOptionPane;
4
5  public class Promedio1 {
6
7     public static void main( String args[] )
8     {
9        int total;        // suma de las calificaciones introducidas por el usuario
10       int contadorCalif;  // número de calificación a introducir a continuación
11       int calificación;   // valor de la calificación
12       int promedio;       // promedio de calificaciones
13
14       String cadenaCalif; // calificación introducida por el usuario
15
```

Figura 4.7 Repetición controlada por contador: Problema del promedio de una clase. (Parte 1 de 3.)

http://librosysolucionarios.net

```
16          // fase de inicialización
17          total = 0;          // inicializar total
18          contadorCalif = 1;  // inicializar contador de ciclo
19
20          // fase de procesamiento
21          while ( contadorCalif <= 10 ) {  // iterar 10 veces
22
23             // pedir la entrada y leer la calificación del usuario
24             cadenaCalif = JOptionPane.showInputDialog(
25                "Escriba la calificación como un entero: " );
26
27             // convertir cadenaCalif en int
28             calificacion = Integer.parseInt( cadenaCalif );
29
30             total = total + calificacion;      // sumar calificación al total
31             contadorCalif = contadorCalif + 1; // incrementar el contador
32
33          } // fin de instrucción while
34
35          // fase de terminación
36          promedio = total / 10;  // división de enteros
37
38          // mostrar el promedio de las calificaciones del examen
39          JOptionPane.showMessageDialog( null, "El promedio de la clase es " + promedio,
40             "Promedio de la clase", JOptionPane.INFORMATION_MESSAGE );
41
42          System.exit( 0 );  // terminar el programa
43
44       } // fin de main
45
46    } // fin de la clase Promedio1
```

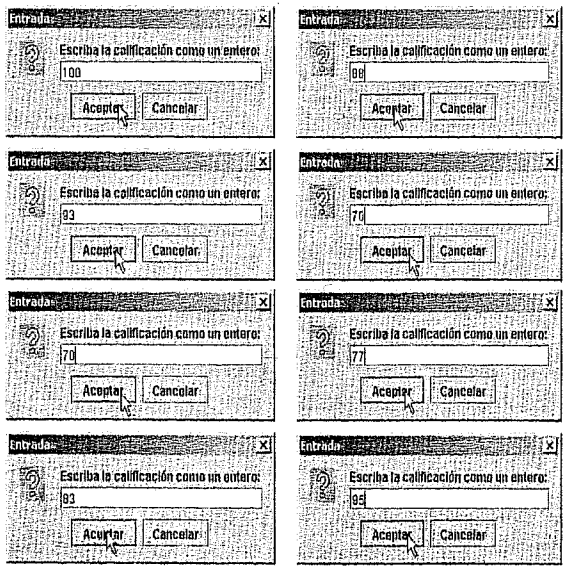


Figura 4.7 Repetición controlada por contador: Problema del promedio de una clase. (Parte 2 de 3.)

http://librosysolucionarios.net

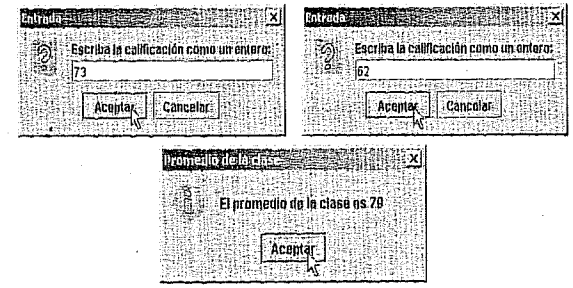


Figura 4.7 Repetición controlada por contador: Problema del promedio de una clase. (Parte 3 de 3.)

Buena práctica de programación 4.5

Si usted prefiere colocar las declaraciones al principio de un método, separe las declaraciones de las instrucciones ejecutables en ese método con una línea en blanco, para resaltar en dónde terminan las declaraciones y en dónde empiezan las instrucciones ejecutables.

Buena práctica de programación 4.6

Declare cada variable en una línea separada, para que los programas sean más legibles.

Buena práctica de programación 4.7

Coloque siempre una línea en blanco antes de una declaración que aparezca entre instrucciones ejecutables, para hacer que la declaración sobresalga en el programa. Esto contribuye a que el programa sea más claro.

Las asignaciones (en las líneas 17 y 18) inicializan `total` en `0` y `contadorCalif` en `1`. Observe que estas inicializaciones ocurren antes de utilizar las variables en los cálculos.

Error común de programación 4.6

Intentar utilizar una variable local antes de inicializarla produce un error de compilación, el cual indica que la variable tal vez no esté inicializada. El valor de una variable local no puede usarse en una expresión sino hasta que esa variable haya sido inicializada. El programa no compilará correctamente hasta que la variable reciba un valor inicial.

Tip para prevenir errores 4.1

Inicialice los contadores y los totales.

La línea 21 indica que la instrucción `while` debe continuar iterando (lo que también se conoce como *hacer un ciclo*) mientras que el valor de `contadorCalif` sea menor o igual a 10. Mientras esta condición sea verdadera, la instrucción `while` ejecutará las instrucciones entre las llaves que delimitan su cuerpo (líneas 21 a 33).

La instrucción en las líneas 24 y 25, que muestra un cuadro de diálogo con el indicador `"Escriba la calificación como un entero:"`, corresponde a la instrucción en seudocódigo *"Introducir la siguiente calificación"*.

Una vez que el usuario escribe un valor para `cadenaCalif`, el programa lo convierte en un valor `int`, en la línea 28. Recuerde de la sección 2.5 que la clase `Integer` es del paquete `java.lang`, que todos los programas en Java importan en forma implícita. El seudocódigo para el problema del promedio de la clase no refleja la instrucción de la línea 28. La instrucción en seudocódigo *"Introducir la siguiente calificación"* requiere que el programador implemente el código que obtiene el valor del usuario y lo convierta en un tipo que pueda utilizarse en el cálculo del promedio. A medida que aprenda a programar, descubrirá que se requieren menos instrucciones de seudocódigo para ayudarle a implementar un programa.

A continuación, la línea 30 actualiza el `total` con la nueva `calificacion` introducida por el usuario. La línea 30 suma `calificacion` al valor anterior de `total` y asigna el resultado a `total`.

La línea 31 suma 1 a `contadorCalif` para indicar que el programa ha procesado una calificación y está listo para recibir la siguiente calificación por parte del usuario. El proceso de incrementar `contadorCalif`

eventualmente hará que `contadorCalif` pase de 10, lo cual hace que la condición en el ciclo `while` se vuelva falsa y el ciclo termine.

Cuando el ciclo termina, la línea 36 asigna a la variable `promedio` el resultado del cálculo del promedio. Las líneas 39 y 40 muestran un cuadro de diálogo que contiene la cadena `"El promedio de la clase es "` seguida por el valor de la variable `promedio`. La cadena `"Promedio de la clase"` (el tercer argumento) es el título del cuadro de diálogo. La línea 44 termina la aplicación.

Después de compilar la declaración de la clase con `javac`, ejecute la aplicación desde la ventana de comandos, utilizando el comando

```
java Promedio1
```

el cual invoca al intérprete de Java y le dice que el método `main` de esta aplicación está declarado en la clase `Promedio1`.

El cálculo del promedio en el programa de la figura 4.7 produjo un resultado entero. En realidad, la suma de los valores de los puntos de calificación en la ejecución de ejemplo es de 796 que, al dividirse entre 10, debe producir el número de punto flotante 79.6 como resultado. En la siguiente sección veremos cómo utilizar enteros en un cálculo de punto flotante.

4.9 Cómo formular algoritmos mediante el mejoramiento de arriba a abajo, paso a paso: Ejemplo práctico 2 (repetición controlada por un centinela)

Generalicemos el problema para los promedios de una clase. Considere el siguiente problema:

> *Desarrollar un programa que calcule el promedio de una clase y que procese las calificaciones para un número arbitrario de estudiantes cada vez que se ejecute.*

En el ejemplo anterior del promedio de una clase, el enunciado del problema especificó el número de estudiantes (10) por adelantado. En este ejemplo no se indica cuántas calificaciones va a introducir el usuario durante la ejecución del programa. El programa debe procesar un número arbitrario de calificaciones. ¿Cómo puede el programa determinar cuándo terminar de introducir calificaciones? ¿Cómo sabrá cuándo calcular e imprimir el promedio de la clase?

Una manera de resolver este problema es utilizar un valor especial denominado *valor centinela* (también llamado *valor de señal*, *valor de prueba* o *bandera*) para indicar el "fin de la introducción de datos". El usuario escribe calificaciones hasta que se haya introducido el número correcto de ellas. Después, el usuario escribe el valor centinela para indicar que no se van a introducir más calificaciones. A la repetición controlada por centinela a menudo se le llama *repetición indefinida*, ya que el número de repeticiones no se conoce antes de que comience la ejecución del ciclo.

Evidentemente, el valor centinela debe elegirse de tal forma que no pueda confundirse con un valor de entrada permitido. Por lo general, las calificaciones de un examen son enteros positivos, por lo que –1 es un valor centinela aceptable para este problema. Por lo tanto, una ejecución del programa para promediar una clase podría procesar una cadena de entradas como 95, 96, 75, 74, 89 y –1. El programa entonces calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89. Observe que –1 es el valor centinela, por lo que no debe entrar en el cálculo del promedio.



Error común de programación 4.7

Seleccionar un valor centinela que sea también un valor de datos permitido es un error lógico.

Método de mejoramiento de arriba a abajo, paso a paso

Desarrollamos el programa para promediar clases con una técnica llamada *mejoramiento de arriba a abajo, paso a paso*, la cual es esencial para el desarrollo de programas bien estructurados. Comenzamos con una representación en seudocódigo de la *cima*, una sola instrucción que transmite la función del programa en general:

> *Determinar el promedio de la clase para el examen*

La *cima* es, en efecto, la representación completa de un programa. Desafortunadamente, la cima rara vez transmite los detalles suficientes como para escribir un programa en Java. Por lo tanto, ahora comenzaremos el

proceso de mejora. Dividiremos la cima en una serie de tareas más pequeñas y las enlistaremos en el orden en el que se van a realizar. Esto arroja como resultado la siguiente *primera mejora*:

> *Inicializar variables*
> *Introducir, sumar y contar las calificaciones del examen*
> *Calcular e imprimir el promedio de la clase*

Esta mejora utiliza sólo la estructura de secuencia; los pasos aquí mostrados deben ejecutarse en orden, uno después del otro.



Observación de ingeniería de software 4.3

Cada mejora, así como la cima en sí, es una especificación completa del algoritmo; sólo varía el nivel del detalle.



Observación de ingeniería de software 4.4

Muchos programas pueden dividirse lógicamente en tres fases: fase de inicialización, en la que se inicializan las variables; fase de procesamiento, en la que se introducen los valores de los datos y se ajustan las variables del programa según sea necesario; y fase de terminación, que calcula y produce los resultados finales.

La observación de ingeniería de software es a menudo todo lo que usted necesita para la primera mejora en el proceso de arriba a abajo. Para avanzar al siguiente nivel de mejora, es decir, la *segunda mejora*, nos comprometemos a usar variables específicas. En este ejemplo necesitamos el total actual de los números, una cuenta de cuántos números se han procesado, una variable para recibir el valor de cada calificación, a medida que el usuario las vaya introduciendo, y una variable para almacenar el promedio calculado. La instrucción en seudocódigo

> *Inicializar las variables*

puede mejorarse como sigue:

> *Inicializar total en cero*
> *Inicializar contador en cero*

Sólo las variables *total* y *contador* necesitan inicializase antes de que puedan utilizarse; las variables *promedio* y *calificacion* (para el promedio calculado y la entrada del usuario, respectivamente) no necesitan inicializarse, ya que sus valores se reemplazarán a medida que se calculen o introduzcan.

La instrucción en seudocódigo

> *Introducir, sumar y contar las calificaciones del examen*

requiere una estructura de repetición (es decir, un ciclo) que introduzca correctamente cada calificación. No sabemos de antemano cuántas calificaciones van a procesarse, por lo que utilizaremos la repetición controlada por centinela. El usuario introduce las calificaciones reales una por una. Después de introducir la última calificación real, el usuario introduce el valor centinela. El programa evalúa el valor centinela después de la introducción de cada valor, y termina el ciclo cuando el usuario introduce el valor centinela. Entonces, la segunda mejora de la instrucción anterior en seudocódigo sería

> *Introducir la primera calificación (puede ser el centinela)*
> *Mientras el usuario no haya introducido aún el centinela*
> *Sumar esta calificación al total actual*
> *Sumar uno al contador de calificaciones*
> *Introducir la siguiente calificación (puede ser el centinela)*

En seudocódigo no utilizamos llaves alrededor del conjunto de instrucciones que forman el cuerpo de la estructura *Mientras*. Simplemente aplicamos sangría a las instrucciones bajo el *Mientras* para mostrar que pertenecen a esta instrucción. De nuevo, el seudocódigo es solamente una herramienta informal para desarrollar programas.

La instrucción en seudocódigo:

> *Calcular e imprimir el promedio de la clase*

http://librosysolucionarios.net

puede mejorarse de la siguiente manera:

Si el contador no es igual a cero
Asignar al promedio el total dividido entre el contador
Imprimir el promedio
De lo contrario
Imprimir "No se introdujeron calificaciones"

Aquí tenemos cuidado de evaluar la posibilidad de una división entre cero; ya que esto es un error lógico que, si no se detecta, haría que el programa fallara o produjera resultados inválidos. La segunda mejora completa del seudocódigo para el problema del promedio de una clase se muestra en la figura 4.8.

Tip para prevenir errores 4.2



Al realizar una división entre una expresión cuyo valor pudiera ser cero, debe evaluar explícitamente esta posibilidad y manejarla de manera apropiada en su programa (como imprimir un mensaje de error), en vez de permitir que ocurra el error.

En las figuras 4.6 y 4.8 incluimos algunas líneas completamente en blanco y sangría en el seudocódigo para facilitar su lectura. Las líneas en blanco separan los algoritmos en seudocódigo en sus diversas fases, y la sangría enfatiza los cuerpos de las estructuras de control.

El algoritmo en seudocódigo en la figura 4.8 resuelve el problema general para promediar una clase. Este algoritmo se desarrolló después de aplicar dos niveles de mejoramiento. En ocasiones, se requieren más niveles de mejoramiento.

Observación de ingeniería de software 4.5



El proceso de mejoramiento de arriba a abajo, paso a paso, finalizará cuando el algoritmo en seudocódigo se especifique con el detalle suficiente como para poder convertir el seudocódigo en Java. Por lo general, la implementación del programa en Java después de esto es mucho más sencilla.

Observación de ingeniería de software 4.6



Algunos programadores experimentados escriben programas sin utilizar herramientas de desarrollo de programas como el seudocódigo. Estos programadores sienten que su meta final es resolver el problema en una computadora y que el escribir seudocódigo simplemente retarda la producción de los resultados finales. Aunque este método pudiera funcionar para problemas sencillos y conocidos, tiende a ocasionar graves errores y retrasos en proyectos grandes y complejos.

La figura 4.9 muestra la aplicación de Java que implementa el algoritmo en seudocódigo de la figura 4.8. Aunque cada calificación es un valor entero, existe la probabilidad de que el cálculo del promedio produzca un número con un punto decimal (es decir, un número real). El tipo `int` no puede representar números reales, por lo que este programa utiliza el tipo `double` para manejar números de punto flotante (como en la sección 3.5),

Inicializar total en cero
Inicializar contador en cero

Introducir la primera calificación (puede ser el centinela)
Mientras el usuario no haya introducido aún el centinela
Sumar esta calificación al total actual
Sumar uno al contador de calificaciones
Introducir la siguiente calificación (puede ser el centinela)

Si el contador no es igual a cero
Asignar al promedio el total dividido entre el contador
Imprimir el promedio
De lo contrario
Imprimir "No se introdujeron calificaciones"

Figura 4.8 Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela.

http://librosysolucionarios.net


```
1  // Fig. 4.9: Promedio2.java
2  // Programa del promedio de una clase mediante la repetición controlada por centinela.
3  import java.text.DecimalFormat; // clase para dar formato a los números
4  import javax.swing.JOptionPane;
5
6  public class Promedio2 {
7
8     public static void main( String args[] )
9     {
10       int total;            // suma de las calificaciones
11       int contadorCalif;    // número de calificaciones introducidas
12       int calificacion;     // valor de la calificación
13
14       double promedio;   // número con punto decimal para el promedio
15
16       String cadenaCalif;  // calificación introducida por el usuario
17
18       // fase de inicialización
19       total = 0;          // inicializar el total
20       contadorCalif = 0;  // inicializar contador de ciclo
21
22       // fase de procesamiento
23       // obtener la primera calificación del usuario
24       cadenaCalif = JOptionPane.showInputDialog(
25          "Escriba calificación como entero o -1 para salir:" );
26
27       // convertir cadenaCalif en int
28       calificacion = Integer.parseInt( cadenaCalif );
29
30       // iterar hasta que el usuario introduzca el valor centinela
31       while ( calificacion != -1 ) {
32          total = total + calificacion;          // sumar calificacion al total
33          contadorCalif = contadorCalif + 1;  // incrementar el contador
34
35          // obtener siguiente calificación del usuario
36          cadenaCalif = JOptionPane.showInputDialog(
37             "Escriba calificación como entero o -1 para salir:" );
38
39          // convertir cadenaCalif en int
40          calificacion = Integer.parseInt( cadenaCalif );
41
42       } // fin de instrucción while
43
44       // fase de terminación
45       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
46
47       // si el usuario introdujo al menos una calificación...
48       if ( contadorCalif != 0 ) {
49
50          // calcular el promedio de todas las calificaciones introducidas
51          promedio = (double) total / contadorCalif;
52
53          // mostrar el promedio con dos dígitos de precisión
54          JOptionPane.showMessageDialog( null,
55             "El promedio de la clase es " + dosDigitos.format( promedio ),
```

Figura 4.9 Repetición controlada por centinela: programa para promediar una clase. (Parte 1 de 2.)

```
56                "Promedio de la clase", JOptionPane.INFORMATION_MESSAGE );
57
58          } // fin de la parte if de la instrucción if...else
59
60          else // si no se introdujeron calificaciones, imprimir el mensaje apropiado
61             JOptionPane.showMessageDialog( null, "No se introdujeron calificaciones",
62                "Promedio de la clase", JOptionPane.INFORMATION_MESSAGE );
63
64          System.exit( 0 );  // terminar la aplicación
65
66       } // fin de main
67
68    } // fin de la clase Promedio2
```



Figura 4.9 Repetición controlada por centinela: programa para promediar una clase. (Parte 2 de 2.)

e introduce un operador especial llamado *operador de conversión de tipo* para convertir los valores `int` en tipo `double` para usarlos en el cálculo del promedio.

En este ejemplo vemos que las estructuras de control pueden apilarse una encima de otra (en secuencia), al igual que un niño apila bloques de construcción. La instrucción `while` (líneas 31 a 42) va seguida por una instrucción `if...else` (líneas 48 a 62) en secuencia. La mayor parte del código en este programa es igual al código de la figura 4.7, por lo que al describir este programa nos concentraremos en las nuevas características y cuestiones.

La línea 14 declara la variable `promedio` de tipo `double`. Esta variable nos permite guardar la clase promedio como un número de punto flotante. La línea 20 inicializa `contadorCalif` en 0, ya que todavía no se han introducido calificaciones. Recuerde que este programa utiliza la repetición controlada por centinela. Para mantener un registro preciso del número de calificaciones introducidas, el programa incrementa `contadorCalif` sólo cuando el usuario introduce un valor permitido para la calificación.

Compare la lógica del programa para la repetición controlada por centinela con la del programa con repetición controlada por contador en la figura 4.7. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 21 a 33 de la figura 4.7) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee y convierte un valor (líneas 24 a 28 de la figura 4.9) antes de llegar al `while`. Este valor determina si el flujo de control del programa debe entrar al cuerpo del `while`. Si la condición del `while` es falsa, el usuario introdujo el valor centinela, por lo que el cuerpo del `while` no se ejecuta (es decir, no se introdujeron calificaciones). Si, por otro lado, la condición es verdadera, el cuerpo comienza a ejecutarse y el ciclo suma al `total` el valor introducido por el usuario. Una vez [illegible] procesado, las líneas 36 a 40 en el cuerpo del ciclo introducen el siguiente valor del usuario, antes de que el programa llegue al final del bloque. A medida que el control del programa se acerca a la llave derecha de terminación (}) del cuerpo en la línea 42, la ejecución continúa con la siguiente evaluación de la condición del `while` (línea 31). La condición utiliza el nuevo valor que acaba de introducir el usuario, para determinar si el cuerpo de la instrucción `while` debe ejecutarse otra vez. Observe que el siguiente valor siempre lo introduce el usuario inmediatamente antes de que el programa evalúe la condición del `while`. Esto permite al programa determinar si el valor que acaba de introducir el usuario es el valor centinela, *antes* de que el programa procese ese valor (es decir, que lo sume al `total`). Si el valor introducido es el valor centinela, el `while` termina y el programa no suma –1 al `total`.

Error común de programación 4.8

Olvidar leer el primer valor de datos antes de una instrucción while que utiliza repetición controlada por centinela, produce errores lógicos que podrían ocasionar que el ciclo se evitara por completo, o podría tratar el valor centinela como un valor de datos permitido.

Buena práctica de programación 4.8

En un ciclo controlado por centinela, los indicadores que solicitan la introducción de datos deben recordar explícitamente al usuario el valor que representa al centinela.

Observe el bloque en el ciclo `while` de la figura 4.9. Sin las llaves, las últimas tres instrucciones en el cuerpo del ciclo quedarían fuera del mismo, ocasionando que la computadora interpretara el código incorrectamente, como se muestra a continuación:

```
while ( calificacion != -1 )
   total = total + calificacion;        // sumar calificación al total
contadorCalif = contadorCalif + 1;  // incrementar el contador

// obtener siguiente calificación del usuario
cadenaCalif = JOptionPane.showInputDialog(
   "Escriba calificación como entero o -1 para salir:" );

// convertir cadenaCalif en int
   calificacion = Integer.parseInt( cadenaCalif );
```

El código anterior ocasionaría un ciclo infinito si el usuario no introduce el centinela `-1` como valor de entrada en las líneas 24 y 25 (antes de la instrucción `while`) en el programa.

Error común de programación 4.9

Omitir las llaves que delimitan a un bloque puede provocar errores lógicos como ciclos infinitos. Para prevenir este problema, algunos programadores encierran el cuerpo de todas las instrucciones de control con llaves.

La línea 45 declara `dosDigitos` como una referencia a un objeto de la clase `DecimalFormat` (paquete `java.text`). Los objetos `DecimalFormat` dan formato a los números. En este ejemplo, decidimos mostrar el promedio de la clase con dos dígitos a la derecha del punto decimal (es decir, redondeado a la centésima más cercana). La línea 45 crea un objeto `DecimalFormat` que se inicializa con el patrón `"0.00"`. Cada `0` especifica una posición de dígito requerida en el número de punto flotante con formato. Este formato específico indica que todos los números con el formato de `dosDigitos` tendrán cuando menos un dígito a la izquierda, y exactamente dos a la derecha del punto decimal. Si el número no cumple con los requerimientos de formato, se insertan ceros en donde sea necesario para cumplir con el formato requerido. La *palabra clave* `new` comienza una *expresión de creación de instancia de una clase*, la cual crea un objeto del tipo especificado a la derecha de `new`. El proceso de crear nuevos objetos se conoce también como *crear una instancia de*, o *instanciar un objeto*. El valor entre paréntesis después del tipo en la expresión de creación de la instancia de una clase se utiliza para *inicializar* (es decir, dar un valor a) el nuevo objeto. A la referencia `dosDigitos` se le asigna el valor de la operación `new`, que es una referencia al nuevo objeto. La instrucción en la línea 45 se lee como "`dosDigitos` *obtiene* el valor de `new DecimalFormat( "0.00" )`".

Observación de ingeniería de software 4.7

Por lo general, los objetos se crean con `new`. Una excepción a esto es una literal de cadena que está contenida en comillas, como `"hola"`. Las literales de cadena son referencias a objetos de la clase `String`, los cuales se crean implícitamente cuando Java encuentra una literal de cadena.

http://librosysolucionarios.net

Los promedios no siempre se evalúan como valores enteros. A menudo, un promedio es un valor que contiene una parte fraccionaria, como 3.333 o 2.7. Estos valores se conocen como *números de punto flotante* y se representan mediante el tipo `double` (vea la sección 3.5). En la línea 14, la variable `promedio` se declara como de tipo `double` para capturar el resultado fraccionario de nuestro cálculo. Sin embargo, el resultado del cálculo `total / contadorCalif` (línea 51) es un entero, ya que `total` y `contadorCalif` son variables enteras. Dividir dos enteros se conoce como *división de enteros*: cualquier parte fraccionaria del cálculo se pierde (es decir, se *trunca*). La parte fraccionaria se pierde antes de que el resultado pueda asignarse a `promedio`, ya que el cálculo se realiza antes de que ocurra la asignación.



Error común de programación 4.10

Suponer que la división de enteros redondea (en lugar de truncar) puede producir resultados incorrectos.

Para realizar un cálculo de punto flotante a partir de valores enteros debemos crear, para el cálculo, valores temporales que sean números de punto flotante. Java cuenta con el *operador unario de conversión de tipo* para llevar a cabo esta tarea. La línea 51 utiliza el operador unario de conversión de tipo `(double)` para crear una copia de punto flotante temporal de su operando, `total` (que aparece a la derecha del operador). Utilizar un operador de conversión de tipo de esta forma es un proceso que se denomina *conversión explícita*. El valor almacenado en `total` sigue siendo un entero. El cálculo ahora consiste de un valor de punto flotante (la versión temporal `double` de `total`) dividido entre el entero `contadorCalif`. Java sabe cómo evaluar sólo expresiones aritméticas en las que los tipos de los operandos sean idénticos. Para asegurar que los operandos sean del mismo tipo, Java realiza una operación llamada *promoción* (o *conversión implícita*) en los operandos seleccionados. Por ejemplo, en una expresión que contenga valores de los tipos `int` y `double`, los valores `int` son *promovidos* a valores `double` para utilizarlos en la expresión. En este ejemplo, Java promueve el valor de `contadorCalif` al tipo `double`, después el programa realiza el cálculo y asigna el resultado de la división de punto flotante a `promedio`. Más adelante en el capítulo, hablaremos sobre todos los tipos estándares y sus reglas de promoción.



Error común de programación 4.11

El operador de conversión de tipo puede utilizarse para convertir entre los tipos numéricos primitivos y para convertir entre los tipos de referencia relacionados (como lo describiremos en el capítulo 9). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.



Buena práctica de programación 4.9

No compare valores de punto flotante para ver su igualdad o desigualdad. En vez de eso, determine si el valor absoluto de la diferencia entre los números es menor que un valor pequeño especificado.

Los operadores de conversión de tipo están disponibles para cualquier tipo. El operador de conversión se forma colocando paréntesis alrededor del nombre de un tipo. Este operador es un *operador unario* (es decir, un operador que utiliza sólo un operando). En el capítulo 2 estudiamos los operadores aritméticos binarios. Java también soporta las versiones unarias de los operadores de suma (`+`) y resta (`-`), por lo que el programador puede escribir expresiones como `-7` o `+5`. Los operadores de conversión de tipo se asocian de derecha a izquierda y tienen la misma precedencia que los demás operadores unarios, como `+` y `-`. Esta precedencia es un nivel mayor que la de los *operadores de multiplicación* `*`, `/` y `%`, y un nivel menor que la de los paréntesis. (Consulte la tabla de precedencia de operadores en el apéndice A.) En nuestras tablas de precedencia, indicamos el operador de conversión de tipos con la notación *(tipo)* para indicar que puede usarse cualquier nombre de tipo para formar un operador de conversión de tipo.

Aunque los números de punto flotante no siempre son 100% precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 36.7, no necesitamos tener una precisión de un gran número de dígitos. Cuando leemos la temperatura en un termómetro como 36.7, en realidad podría ser 36.6999473210643. Considerar este número simplemente como 36.7 está bien para la mayoría de las aplicaciones.

Otra manera en la que se desarrollan los números de punto flotante es a través de la división. Cuando dividimos 10 entre 3, el resultado es 3.3333333... donde la secuencia de números se repite de manera infinita. La computadora asigna sólo una cantidad fija de espacio para guardar ese valor, por lo que evidentemente el valor de punto flotante almacenado es sólo una aproximación.

http://librosysolucionarios.net



Error común de programación 4.12

Usar números de punto flotante suponiendo que se representan de manera exacta puede producir resultados imprecisos.

4.10 Cómo formular algoritmos mediante el mejoramiento de arriba a abajo, paso a paso: Ejemplo práctico 3 (estructuras de control anidadas)

En el siguiente ejemplo formularemos una vez más el algoritmo utilizando seudocódigo y el mejoramiento de arriba a abajo, paso a paso, y después escribiremos el correspondiente programa en Java. Hemos visto que las estructuras de control pueden apilarse una encima de otra (en secuencia), de la misma forma en que un niño apila bloques de construcción. En este ejemplo práctico examinaremos la otra forma en la que pueden conectarse las instrucciones de control, a saber, mediante el *anidamiento* de una instrucción de control dentro de otra.

Considere el siguiente enunciado de un problema:

> *Una universidad ofrece un curso que prepara a los estudiantes para el examen estatal de certificación del estado como corredores de bienes raíces. El año pasado, diez de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron sus estudiantes en el examen. A usted se le ha pedido que escriba un programa para sintetizar los resultados. Se le dio una lista de estos 10 estudiantes; junto a cada nombre hay un 1 escrito, si el estudiante aprobó el examen, o un 2 si lo reprobó.*
>
> *Su programa debe analizar los resultados del examen de la siguiente manera:*
>
> 1. *Introducir cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje "Escriba el resultado" en la pantalla, cada vez que el programa solicite otro resultado de la prueba.*
> 2. *Contar el número de resultados de la prueba, de cada tipo.*
> 3. *Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y el número de estudiantes que reprobaron.*
> 4. *Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje "Aumentar la colegiatura".*

Después de leer el enunciado del programa cuidadosamente, hacemos las siguientes observaciones:

1. El programa debe procesar los resultados de la prueba para 10 estudiantes. Puede usarse un ciclo controlado por contador, ya que el número de resultados de la prueba se conoce de antemano.
2. Cada resultado de la prueba tiene un valor numérico, ya sea 1 o 2. Cada vez que el programa lee un resultado de la prueba, debe determinar si el número es 1 o 2. Nosotros evaluamos un 1 en nuestro algoritmo. Si el número no es 1, suponemos que es un 2. (El ejercicio 4.18 considera las consecuencias de esta suposición.)
3. Dos contadores se utilizan para llevar el registro de los resultados de los exámenes: uno para contar el número de estudiantes que aprobaron el examen y uno para contar el número de estudiantes que reprobaron el examen.
4. Una vez que el programa ha procesado todos los resultados, debe decidir si más de ocho estudiantes aprobaron el examen.

Veamos ahora el mejoramiento de arriba a abajo, paso a paso. Comencemos con la representación del seudocódigo de la cima:

Analizar los resultados del examen y decidir si debe aumentarse la colegiatura o no.

Una vez más, es importante enfatizar que la cima es una representación completa del programa, pero es probable que se necesiten varias mejoras antes de que el seudocódigo pueda evolucionar de manera natural en un programa en Java.

Nuestra primera mejora es

Inicializar variables
Introducir las 10 calificaciones del examen y contar los aprobados y reprobados
Imprimir un resumen de los resultados de los exámenes y decidir si debe aumentarse la colegiatura.

Aquí también, aún cuando tenemos una representación completa del programa, es necesario mejorarla. Ahora nos comprometemos con variables específicas. Se necesitan contadores para registrar los aprobados y reprobados; utilizaremos un contador para controlar el proceso de los ciclos y necesitaremos una variable para guardar la entrada del usuario. La variable en la que se almacenará la entrada del usuario no se inicializa, ya que su valor proviene del usuario durante cada iteración del ciclo.

La instrucción en seudocódigo

Inicializar variables

puede mejorarse de la siguiente manera:

Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en cero

Observe que sólo se inicializan los contadores.

La instrucción en seudocódigo

Introducir las 10 calificaciones del examen y contar los aprobados y reprobados

requiere un ciclo en el que se introduzca sucesivamente el resultado de cada examen. Sabemos de antemano que hay precisamente 10 resultados del examen, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, *anidado* dentro del ciclo), una estructura de selección doble determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, la mejora al seudocódigo anterior es

Mientras el contador de estudiantes sea menor o igual a 10
Introducir el siguiente resultado del examen

Si el estudiante aprobó
Sumar uno a aprobados
De lo contrario
Sumar uno a reprobados

Sumar uno al contador de estudiantes

Nosotros utilizamos líneas en blanco para aislar la estructura de control *Si...De lo contrario*, lo cual mejora la legibilidad.

La instrucción en seudocódigo

Imprimir un resumen de los resultados de los exámenes y decidir si debe aumentarse la colegiatura

puede mejorarse de la siguiente manera:

Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
Imprimir "Aumentar la colegiatura"

La segunda mejora completa aparece en la figura 4.10. Observe que también se utilizan líneas en blanco para separar la estructura *Mientras* y mejorar la legibilidad del programa. Este seudocódigo está ahora lo suficientemente mejorado para su conversión a Java. El programa en Java y una ejecución de ejemplo se muestran en la figura 4.11.

Las líneas 10 a 16 declaran las variables utilizadas en `main` para procesar los resultados del examen. Varias de estas declaraciones utilizan la habilidad de Java para incorporar la inicialización de variables en las declaraciones (a `aprobados` se le asigna `0`, a `reprobados` se le asigna `0` y a `contadorEstudiante` se le asigna `1`). Los programas con ciclos pueden requerir la inicialización al inicio de cada repetición; dicha reinicialización se realiza generalmente mediante instrucciones de asignación.

Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en uno

Mientras el contador de estudiantes sea menor o igual a 10
Introducir el siguiente resultado del examen

Si el estudiante aprobó
Sumar uno a aprobados
De lo contrario
Sumar uno a reprobados
Sumar uno al contador de estudiantes

Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
Imprimir "Aumentar colegiatura"

Figura 4.10 El seudocódigo para el problema de los resultados del examen.

```
1  // Fig. 4.11: Analisis.java
2  // Análisis de los resultados de un examen.
3  import javax.swing.JOptionPane;
4
5  public class Analisis {
6
7     public static void main( String args[])
8     {
9        // inicializar variables en las declaraciones
10       int aprobados = 0;               // número de aprobados
11       int reprobados = 0;              // número de reprobados
12       int contadorEstudiantes = 1;     // contador de estudiantes
13       int resultado;                   // resultado de un examen
14
15       String entrada;                  // valor introducido por el usuario
16       String salida;                   // cadena de salida
17
18       // procesar 10 estudiantes utilizando un ciclo controlado por contador
19       while ( contadorEstudiantes <= 10 ) {
20
21          // pedir al usuario la entrada y obtener el valor del usuario
22          entrada = JOptionPane.showInputDialog(
23             "Escriba el resultado (1 = aprobado, 2 = reprobado)" );
24
25          // convertir resultado en int
26          resultado = Integer.parseInt( entrada );
27
28          // si resultado es 1, incrementar aprobados; if...else anidado en while
29          if ( resultado == 1 )
30             aprobados = aprobados + 1;
31
32          else // si resultado no es 1, incrementar reprobados
33          reprobados = reprobados + 1;
```

Figura 4.11 Estructuras de control anidadas: problema de los resultados del examen. (Parte 1 de 2.)

http://librosysolucionarios.net

```
34
35          // incrementar contadorEstudiantes para que el ciclo termine eventualmente
36          contadorEstudiantes = contadorEstudiantes + 1;
37
38       } // fin de instrucción while
39
40       // fase de terminación; preparar y mostrar resultados
41       salida = "Aprobados: " + aprobados + "\nReprobados: " + reprobados;
42
43       // determinar si aprobaron más de 8 estudiantes
44       if ( aprobados > 8 )
45          salida = salida + "\nAumentar colegiatura";
46
47       JOptionPane.showMessageDialog( null, salida,
48          "Análisis de resultados del examen",
49          JOptionPane.INFORMATION_MESSAGE );
50
51       System.exit( 0 );  // terminar la aplicación
52
53    } // fin de main
54
55 } // fin de la clase Analisis
```

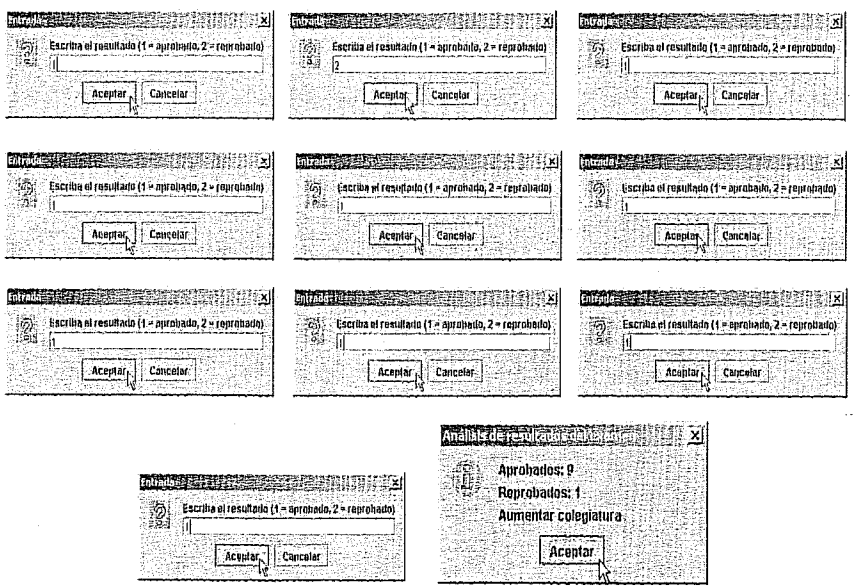


Figura 4.11 Estructuras de control anidadas: problema de los resultados del examen. (Parte 2 de 2.)

Buena práctica de programación 4.10

Inicializar las variables locales cuando se declaran ayuda a evitar mensajes de advertencia del compilador por datos no inicializados.

Observe que la instrucción `if...else` de las líneas 29 a 33 está anidada en la instrucción `while` (líneas 19 a 38). Note el uso de la referencia `output` de la clase `String` en las líneas 41 y 45 para crear la cadena que las líneas 47 a 49 muestran en un cuadro de diálogo de mensaje.

http://librosysolucionarios.net

4.11 Operadores de asignación compuestos

Java cuenta con varios operadores de asignación compuestos para abreviar las expresiones de asignación. Cualquier instrucción de la forma

variable = variable operador expresión;

en donde *operador* es uno de los operadores binarios +, -, *, / o % (o alguno de los otros que veremos más adelante en el libro), puede escribirse de la siguiente forma:

variable operador= expresión;

Por ejemplo, puede abreviar la instrucción

```
c = c + 3;
```

mediante el *operador de asignación compuesto de suma*, +=, de la siguiente manera:

```
c += 3;
```

El operador += suma el valor de la expresión que está a la derecha del operador, al valor de la variable que está a la izquierda del operador, y almacena el resultado en la variable que está a la izquierda del operador. Por lo tanto, la expresión de asignación `c += 3` suma `3` a `c`. La figura 4.12 muestra los operadores de asignación compuestos aritméticos, algunas expresiones de ejemplo en las que se utilizan los operadores y las explicaciones de lo que estos operadores hacen.

4.12 Operadores de incremento y decremento

Java proporciona dos operadores unarios para sumar 1, o restar 1, al valor de una variable numérica. Estos operadores son el *operador unario de incremento*, ++, y el *operador unario de decremento*, --, los cuales se sintetizan en la figura 4.13. Un programa puede incrementar en 1 el valor de una variable llamada `c`, utilizando el operador de incremento, ++, en lugar de usar la expresión `c = c + 1` o `c += 1`. Al operador de incremento o decremento que se coloca antes de una variable se le llama *operador de preincremento o predecremento*, respectivamente. Al operador de incremento o decremento que se coloca después de una variable se le llama *operador de postincremento* o *postdecremento*, respectivamente.[7]

Al preincrementar (o predecrementar) una variable, ésta se incrementa (o decrementa) en 1, y después el nuevo valor de la variable se utiliza en la expresión en la que aparece. Al postincrementar (o postdecrementar) una variable, el valor actual de la variable se utiliza en la expresión en la que aparece y después el valor de la variable se incrementa (o decrementa) en 1.

Buena práctica de programación 4.11

A diferencia de los operadores binarios, los operadores unarios deben colocarse enseguida de sus operandos, sin espacios entre ellos.

Operador de asignación	Expresión de ejemplo	Explicación	Asigna
Suponer que: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10 a c`
`-=`	`d -= 4`	`d = d - 4`	`1 a d`
`*=`	`e *= 5`	`e = e * 5`	`20 a e`
`/=`	`f /= 3`	`f = f / 3`	`2 a f`
`%=`	`g %= 9`	`g = g % 9`	`3 a g`

Figura 4.12 Operadores de asignación aritméticos.

7. La especificación del lenguaje Java utiliza los términos *operador de incremento prefijo, operador de incremento postfijo, operador de decremento prefijo* y *operador de decremento postfijo* para describir a estos operadores.

Operador	Llamado	Expresión de ejemplo	Explicación
++	preincremento	++a	Incrementar a en 1, después utilizar el nuevo valor de a en la expresión en que esta variable reside.
++	postincremento	a++	Usar el valor actual de a en la expresión en la que esta variable reside, después incrementar a en 1.
--	predecremento	--b	Decrementar b en 1, después utilizar el nuevo valor de b en la expresión en que esta variable reside.
--	postdecremento	b--	Usar el valor actual de b en la expresión en la que esta variable reside, después decrementar b en 1.

Figura 4.13 Los operadores de incremento y decremento.

La aplicación de la figura 4.14 demuestra la diferencia entre la versión de preincremento y la versión de predecremento del operador de incremento ++. Al postincrementar la variable c, ésta se incrementa después de ser usada en la llamada al método `System.out.println` (línea 13). Al preincrementar la variable c, ésta se incrementa antes de ser usada en la llamada al método `System.out.println` (línea 21).

```
1  // Fig. 4.14: Incremento.java
2  // Los operadores preincremento y postdecremento.
3
4  public class Incremento {
5
6     public static void main( String args[])
7     {
8        int c;
9
10       // demostrar el uso de postincremento
11       c = 5;                      // asignar 5 a c
12       System.out.println( c );    // imprimir 5
13       System.out.println( c++ );  // imprimir 5 y después postincremento
14       System.out.println( c );    // imprimir 6
15
16       System.out.println();       // saltar una línea
17
18       // demostrar el uso de preincremento
19       c = 5;                      // asignar 5 a c
20       System.out.println( c );    // imprimir 5
21       System.out.println( ++c );  // preincrementar y después imprimir 6
22       System.out.println( c );    // imprimir 6
23
24    } // fin de main
25
26 } // fin de la clase Incremento
```

```
5
5
6

5
6
6
```

Figura 4.14 Preincrementar y postincrementar.

http://librosysolucionarios.net

El programa muestra el valor de c antes y después de utilizar el operador ++. El operador de decremento (--) funciona de manera similar.

La línea 16 utiliza a `System.out.println` para imprimir una línea en blanco. Si `println` no recibe argumentos, simplemente imprime un carácter de nueva línea.

Los operadores de asignación compuestos aritméticos y los operadores de incremento y decremento pueden utilizarse para simplificar las instrucciones de los programas. Por ejemplo, las tres instrucciones de asignación de la figura 4.11 (líneas 30, 33 y 36)

```
aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;
```

pueden escribirse en forma más concisa con operadores de asignación, de la siguiente manera:

```
aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;
```

con operadores de preincremento de la siguiente forma:

```
++aprobados;
++reprobados;
++contadorEstudiantes;
```

o con operadores de postincremento de la siguiente forma:

```
aprobados++;
reprobados++;
contadorEstudiantes++;
```

Es muy importante tener en cuenta que, al incrementar o decrementar una variable que se encuentre en una instrucción por sí sola, las formas preincremento y postincremento tienen el mismo efecto, al igual que las formas predecremento y postdecremento. Solamente cuando una variable aparece en el contexto de una expresión más grande es cuando los operadores preincremento y postdecremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).

Error común de programación 4.13

Sólo una variable o una expresión, que tenga como resultado una variable, pueden aparecer en el lado izquierdo de un operador de asignación. Tratar de usar el operador de incremento o decremento en una expresión a la que no se le pueda asignar un valor es un error de sintaxis. Por ejemplo, escribir ++(x + 1) es un error de sintaxis, ya que (x + 1) no es una variable.

La figura 4.15 muestra la precedencia y la asociatividad de los operadores que se han presentado hasta este punto. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. Observe que el operador condicional (?=), los operadores unarios de incremento (++), decremento (--), suma (+) y resta (-), los operadores de conversión de tipo y los operadores de asignación =, +=, -=, *=, /= y %= se asocian de derecha a izquierda.

Operadores	Asociatividad	Tipo
++ --	derecha a izquierda	postfijo unario
++ -- + - (*tipo*)	derecha a izquierda	unario
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
?:	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación

Figura 4.15 Precedencia y asociatividad de los operadores vistos hasta ahora.

Todos los demás operadores en la tabla de precedencia de operadores de la figura 4.15 se asocian de izquierda a derecha. La tercera columna nombra a los grupos de operadores.

4.13 Tipos primitivos

La tabla de la figura 4.16 muestra los ocho tipos primitivos en Java. Al igual que sus lenguajes antecesores C y C++, Java requiere que todas las variables tengan un tipo. Es por esta razón que Java se conoce como un *lenguaje fuertemente tipificado*.

En los programas de C y C++, los programadores frecuentemente tenían que escribir versiones independientes para dar soporte a las distintas plataformas de computadoras, ya que no se garantizaba que los tipos primitivos fueran idénticos de computadora en computadora. Por ejemplo, un valor `int` en un equipo podría representarse mediante 16 bits (2 bytes) de memoria, mientras que un valor `int` en otro equipo podría representarse mediante 32 bits (4 bytes) de memoria. En Java, los valores `int` siempre son de 32 bits (4 bytes).



Tip de portabilidad 4.1

A diferencia de C y C++, los tipos primitivos en Java son portables en todas las plataformas con soporte para Java. Ésta y muchas otras características de portabilidad de Java permiten a los programadores escribir programas una vez, sin necesidad de saber en cuál plataforma se va a ejecutar el programa. Este atributo se conoce como WORA (Write Once, Run Anywhere; Se escribe una vez, se ejecuta en cualquier lugar).

Tipo	Tamaño en bits	Valores	Estándar
`boolean`		`true` o `false` [*Nota:* La representación de un valor booleano es específica para la Máquina virtual de Java en cada plataforma.]	
`char`	16	`'\u0000'` hasta `'\uFFFF'` (0 hasta 65535)	(Conjunto de caracteres Unicode de ISO)
`byte`	8	−128 hasta +127 (-2^7 hasta $2^7 - 1$)	
`short`	16	−32,768 hasta +32,767 (-2^{15} hasta $2^{15} - 1$)	
`int`	32	−2,147,483,648 hasta +2,147,483,647 (-2^{31} hasta $2^{31} - 1$)	
`long`	64	−9,223,372,036,854,775,808 hasta +9,223,372,036,854,775,807 (-2^{63} hasta $2^{63} - 1$)	
`float`	32	*Rango negativo:* −3.4028234663852886E+38 hasta −1.4012984632481707e-45 *Rango positivo:* 1.4012984632481707e-45 hasta 3.4028234663852886E+38	(IEEE 754, números de punto flotante)
`double`	64	*Rango negativo:* −1.7976931348623157E+308 hasta −4.94065645841246544e-324 *Rango positivo:* 4.94065645841246544e-324 hasta 1.7976931348623157E+308	(IEEE 754, números de punto flotante)

Figura 4.16 Tipos primitivos de Java.

http://librosysolucionarios.net

Cada uno de los tipos de la figura 4.16 se enlista con su tamaño en bits (hay ocho bits en un byte) y su rango de valores. Como los diseñadores de Java desean que sea lo más portable posible, utilizan estándares reconocidos internacionalmente tanto para los formatos de caracteres (Unicode) como para los números de punto flotante (IEEE 754).

Cuando se declaran variables de alguno de los tipos primitivos fuera de un método, se les asignan automáticamente valores predeterminados, a menos que se inicialicen en forma explícita. Las variables de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor `0` de manera predeterminada. Las variables de tipo `boolean` reciben el valor `false` de manera predeterminada.

4.14 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar los atributos de las clases

En la sección 3.7 comenzamos la primera fase de un diseño orientado a objetos (DOO) para nuestro simulador de elevador; analizamos el enunciado del problema e identificamos las clases necesarias para implementar el simulador. Enlistamos los sustantivos en el enunciado del problema e identificamos una clase separada para cada categoría de sustantivo y frase nominal que tuviera significado para la simulación del elevador. Después modelamos las clases y sus relaciones en un diagrama de clases de UML (figura 3.19). Las clases tienen atributos (datos) y operaciones (comportamientos). Los atributos de una clase se implementan como campos en los programas en Java; los comportamientos de una clase se implementan como métodos. En esta sección determinaremos muchos de los atributos necesarios en el simulador del elevador. En el capítulo 5 examinaremos cómo estos atributos representan el *estado*, o condición, de un objeto. En el capítulo 6 determinaremos el comportamiento de las clases.

Considere los atributos de algunos objetos reales: los atributos de una persona incluyen la altura y el peso. Los atributos de un radio incluyen el ajuste de la estación, del volumen y de la banda AM o FM. Los atributos de un auto incluyen las lecturas del velocímetro y del odómetro, la cantidad de gasolina en el tanque, la velocidad (engranaje) en la que se encuentra el motor, etc. Los atributos de una computadora personal incluyen al fabricante (por ejemplo, Dell, Sun, Apple, IBM, etc.), el tipo de monitor (por ejemplo, plano o CRT), el tamaño de la memoria principal (en megabytes), el tamaño del disco duro (en gigabytes), etcétera.

Para identificar los atributos de las clases en nuestro sistema debemos buscar palabras y frases descriptivas en el enunciado del problema. Por cada palabra o frase descriptiva que encontremos, debemos crear un atributo y asignarlo a una de las clases identificadas en la sección 3.7. También creamos atributos para representar cualquier dato adicional que pueda necesitar una clase (a medida que la necesidad de estos datos se vuelva más clara, a lo largo del proceso de diseño).

Examinemos el enunciado del problema para descubrir los diversos atributos de cada clase. La figura 4.17 enlista las palabras o frases del enunciado del problema que describen a cada una de las clases.

Clase	Palabras y frases descriptivas
`ConductoElevador`	[sin palabras ni frases descriptivas]
`Elevador`	se desplaza
	es llamado
	piso actual
	piso de destino
	capacidad de sólo una persona
	cinco segundos para viajar de un piso al otro
`Persona`	única
	espera / se mueve
	piso actual

Figura 4.17 Palabras y frases descriptivas del enunciado del problema. (Parte 1 de 2.)

Clase	Palabras y frases descriptivas
`Piso`	primero o segundo; capacidad para sólo una persona
`BotonPiso`	oprimido / restablecido
`BotonElevador`	oprimido / restablecido
`PuertaPiso`	puerta abierta / puerta cerrada
`PuertaElevador`	puerta cerrada / puerta abierta
`Timbre`	[sin palabras ni frases descriptivas]
`Luz`	encendida / apagada

Figura 4.17 Palabras y frases descriptivas del enunciado del problema. (Parte 2 de 2.)

La clase `Elevador` tiene varios atributos. Las frases "se desplaza" y "es llamado" describen los posibles estados del `Elevador` (presentaremos los estados en la siguiente sección "Acerca de los objetos"), por lo que incluimos `movimiento` y `llamado` como atributos `boolean`. El `Elevador` también llega a un "piso de destino", por lo que incluimos el atributo `pisoDestino`, el cual representa el `Piso` al que viajará el `Elevador`. Aunque el enunciado del problema no menciona explícitamente que el `Elevador` sale de un `Piso` actual, podemos inferir otro atributo llamado `pisoActual`, que representa el `Piso` en el que está descansando el `Elevador`. El enunciado del problema especifica que "el elevador y los pisos tienen cada uno capacidad de sólo una persona", por lo que incluimos el atributo `capacidad` para la clase `Elevador` (y la clase `Piso`) y establecimos su valor en `1`. Por último, el enunciado del problema especifica que el elevador "tarda cinco segundos en viajar de un piso al otro", por lo que agregamos el atributo `tiempoRecorrido` y establecimos su valor en `5`.

La clase `Persona` tiene varios atributos. El usuario debe poder "crear una persona única", lo cual implica que cada objeto `Persona` debe tener un identificador único. Asignamos el atributo `ID` al objeto `Persona`. El atributo `ID` ayuda a identificar a ese objeto `Persona`. Además, el enunciado del problema especifica que la `Persona` puede estar "esperando a que llegue el elevador". Por lo tanto, "esperar" es un estado al que puede entrar el objeto `Persona`. Aunque no se menciona explícitamente, si la `Persona` no está esperando el `Elevador`, entonces se está moviendo hacia (o alejándose de) el `Elevador`. Asignamos el atributo `movimiento` de tipo `boolean` a la clase `Persona`. Cuando este atributo se establece en `false`, la `Persona` está "esperando". Por último, la frase "en ese piso" implica que la `Persona` ocupa un piso. No podemos asignar una referencia `Piso` a la clase `Persona`, ya que nos interesan solamente los atributos. Sin embargo, queremos incluir la ubicación del objeto `Persona` en el modelo, por lo que incluimos el atributo `pisoActual`, que puede tener el valor `1` o `2`.

La clase `Piso` tiene un atributo `capacidad`. El enunciado del problema especifica que el usuario puede situar a la persona ya sea en el "primer o segundo piso"; por lo tanto, un objeto `Piso` requiere de un valor que lo identifique como el primer o segundo piso, para lo cual incluimos el atributo `numeroPiso`.

De acuerdo con el enunciado del problema, el `BotonElevador` y el `BotonPiso` son "oprimidos" por una `Persona`. Los botones también pueden ser "restablecidos". El estado de cada botón puede ser "oprimido" o "restablecido". Incluimos el atributo `oprimido` tipo `boolean` en ambas clases de botones. Cuando `oprimido` es `true`, el objeto botón está oprimido; cuando `oprimido` es `false`, el objeto botón se restablece. Tanto la puerta del elevador como la puerta del piso están ya sea "abiertas" o "cerradas", por lo que incluimos el atributo `abierta` tipo `Boolean` en las clases `PuertaElevador` y `PuertaPiso`. La clase `Luz` también se encuentra dentro de esta categoría; la luz puede estar "encendida" o "apagada", por lo que incluimos el atributo `luzEncendida` tipo `boolean` en la clase `Luz`. Observe que, aunque el enunciado del problema menciona que el timbre suena, no hay mención de cuándo el timbre "está sonando", por lo que no incluimos un atributo `sonando` separado para la clase `Timbre`. A medida que vayamos progresando a través de este ejemplo práctico, seguiremos agregando, modificando y eliminando información acerca de las clases en nuestro sistema.

El diagrama de clases de la figura 4.18 muestra algunos de los atributos para cada clase del sistema; las palabras y frases descriptivas en la figura 4.17 nos ayudan a generar estos atributos. Observe que, por

http://librosysolucionarios.net

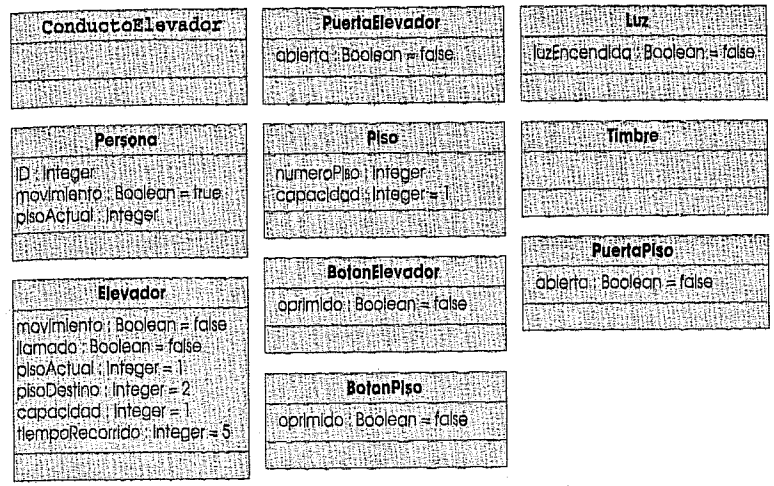


Figura 4.18 Clases con atributos.

cuestión de simpleza, la figura 4.18 no muestra las asociaciones entre los objetos; mostramos estas asociaciones en la figura 3.19. Recuerde que en la sección 3.7 vimos que en UML, los atributos de una clase se colocan en el compartimiento intermedio del rectángulo de la clase.

Considere el atributo `abierta` de la clase `PuertaElevador`:

```
abierta : Boolean = false
```

Este listado contiene tres piezas de información sobre el atributo. El *nombre del atributo* es `abierta`. El *tipo del atributo* es `Boolean`.[8] El tipo depende del lenguaje utilizado para implementar el sistema de software. En Java, por ejemplo, un atributo puede representarse mediante un *tipo primitivo*, como `boolean` o `float`, o mediante un tipo de referencia como una clase; comenzaremos nuestro estudio de las clases en el capítulo 8, en donde veremos que cada nueva clase es un nuevo tipo de referencia.

También podemos indicar un valor inicial para cada atributo. El atributo `abierta` de la clase `PuertaElevador` tiene un valor inicial de `false`. Esto indica que la puerta del elevador está cerrada inicialmente. Si un atributo específico no tiene un valor inicial especificado, sólo se muestra su nombre y su tipo (separados por dos puntos). Por ejemplo, el atributo `ID` de la clase `Persona` es un entero; en Java, el atributo `ID` es de tipo `int`. Aquí no mostramos un valor inicial, ya que el valor de este atributo es un número que no conocemos todavía; este número se determinará en tiempo de ejecución. El atributo entero `pisoActual` de la clase `Persona` también se determina hasta que el programa se ejecute; este atributo se determinará cuando el usuario de la simulación decida en qué `Piso` va a colocar a la `Persona`.

Observe que la figura 4.18 no incluye los atributos para la clase `ConductoElevador`. En realidad, esta clase tiene siete atributos que podemos determinar de las asociaciones modeladas en el diagrama de clases de la figura 3.19: las referencias al objeto `Elevador`, los dos objetos `BotonPiso`, los dos objetos `PuertaPiso` y los dos objetos `Luz`. La clase `Elevador` también contiene tres atributos definidos por el programador: las referencias al objeto `BotonElevador`, el objeto `PuertaElevador` y el objeto `Timbre`. Para

8. Observe que los tipos de los atributos en la figura 4.18 se encuentran en notación de UML. Asociaremos los tipos de atributo `Boolean` e `Integer` en el diagrama de UML con los tipos de atributos `boolean` e `int` en Java, respectivamente. En el capítulo 3 describimos que Java cuenta con una "clase de envoltura de tipos" para cada tipo primitivo. Las clases de envoltura de tipos en Java tienen la misma notación que la de UML para los tipos de atributos; sin embargo, cuando implementemos nuestro diseño en Java empezando en el capítulo 8, utilizaremos tipos primitivos por cuestión de simpleza. Decidir entre usar tipos primitivos o clases de envoltura de tipos es una cuestión específica de la implementación, la cual no debe mencionarse en UML.

ahorrar espacio, no mostraremos estos atributos adicionales en nuestros diagramas de clases; sin embargo, los incluiremos en el código de los apéndices.

El diagrama de clases de la figura 4.18 proporciona una base para la estructura de nuestro modelo, pero el diagrama no está completo. Por ejemplo, el atributo `pisoActual` en la clase `Persona` representa el piso en el cual se ubica actualmente una persona. Sin embargo, ¿en qué piso se encuentra la persona cuando ésta viaja en el elevador? Estos atributos todavía no representan suficientemente bien la estructura del modelo. A medida que presentemos más acerca de UML y del diseño orientado a objetos, continuaremos reforzando la estructura de nuestro modelo.

RESUMEN

- Un algoritmo es un procedimiento para resolver un problema en términos de las acciones a ejecutar, y el orden en el que las acciones deben ejecutarse.
- Al proceso de especificar el orden en el que las instrucciones van a ejecutarse en un programa de computadora se le llama control del programa.
- El seudocódigo ayuda a un programador a "idear" un programa antes de intentar escribirlo en un lenguaje de programación.
- Los diagramas de actividad son parte del Lenguaje de Modelado Unificado (UML), un estándar industrial para modelar sistemas de software.
- Un diagrama de actividades modela el flujo de trabajo (también conocido como la actividad) de un sistema de software.
- Los diagramas de actividad están compuestos de símbolos de propósito especial como los símbolos de estados de acción, rombos y pequeños círculos; estos símbolos se conectan mediante flechas de transición, las cuales representan el flujo de la actividad.
- Al igual que el seudocódigo, los diagramas de actividad ayudan a los programadores a desarrollar y representar algoritmos; aunque muchos programadores prefieren el seudocódigo.
- Un estado de acción se representa mediante líneas horizontales paralelas, conectadas en cada extremo y con arcos convexos. La expresión de acción aparece dentro del estado de acción.
- Las flechas en el diagrama de actividad se llaman flechas de transición. Estas flechas modelan las transiciones, que indican el orden en el que se llevan a cabo los estados de acción.
- El círculo relleno que se encuentra en la parte superior del diagrama de actividad representa el estado inicial: el comienzo del flujo de trabajo antes de que el programa realice la actividad modelada.
- El círculo sólido rodeado por una circunferencia, que aparece en la parte inferior de un diagrama de actividad, representa el estado final: el término del flujo de trabajo después de que el programa realiza la actividad.
- Los rectángulos con las esquinas superiores derechas dobladas se llaman notas en UML. Las notas son comentarios que describen el propósito de los símbolos en el diagrama. Una línea punteada conecta a cada nota con el elemento que ésta describe.
- El rombo, o símbolo de decisión, indica que se va a tomar una decisión; este símbolo indica que el flujo de trabajo continuará a lo largo de una ruta determinada por las condiciones de guardia asociadas, que pueden ser verdaderas o falsas. Cada flecha de transición que sale de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, encima o enseguida de la flecha de transición). Si una condición de guardia específica es verdadera, el flujo de trabajo entra al estado de acción al que apunta esa flecha de transición.
- El rombo también representa al símbolo de fusión, el cual une dos flujos de actividad en uno solo. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar que varios flujos de actividad se fusionan para continuar la actividad. El símbolo de fusión no tiene su contraparte en código de Java.
- El mejoramiento de arriba a abajo, paso a paso, es un proceso para mejorar el seudocódigo, manteniendo una representación completa del programa durante cada mejoramiento.
- Las declaraciones son mensajes para el compilador, que le indican los nombres y atributos de las variables y le dicen que reserve el espacio para ellas.
- Existen tres tipos de estructuras de control: secuencia, selección y repetición.
- La estructura de secuencia está integrada en Java; de manera predeterminada, las instrucciones se ejecutan en el orden en el que aparecen.

- Una estructura de selección elige entre varios cursos alternativos de acción.
- La instrucción `if` ejecuta una acción indicada solamente cuando la condición es verdadera.
- La instrucción `if...else` especifica acciones separadas a ejecutar cuando la condición es verdadera y cuando es falsa.
- Cuando debe ejecutarse más de una instrucción en donde normalmente aparece una sola instrucción, las instrucciones deben ir encerradas entre llaves, formando un bloque. El bloque puede colocarse en cualquier lugar en el que se pueda colocar una sola instrucción.
- Una instrucción vacía, la cual implica que no va a realizarse ninguna acción, se indica mediante un punto y coma (`;`).
- Una estructura de repetición especifica que una acción va a repetirse mientras cierta condición sea verdadera.
- El formato para la instrucción `while` es

  `while` (*condición*)
  instrucción

- La repetición controlada por contador se utiliza cuando el número de iteraciones se conoce de antemano.
- Un número de punto flotante (valor que contiene una parte fraccionaria) se representa mediante los tipos `float` o `double`.
- El operador unario de conversión de tipos (`double`) crea una copia temporal de su operando, como número de punto flotante.
- La repetición controlada por centinela se utiliza cuando el número de iteraciones no se conoce de antemano.
- Una estructura de control anidada aparece en el cuerpo de otra estructura de control.
- Java proporciona los operadores de asignación compuestos aritméticos `+=`, `-=`, `*=`, `/=` y `%=`, que ayudan a abreviar ciertos tipos comunes de expresiones.
- El operador incremento, `++`, y el operador decremento, `--`, incrementan o decrementan una variable en `1`, respectivamente. Si el operador se coloca antes de la variable, ésta se incrementa o decrementa en `1` primero, y después se utiliza en la expresión. Si el operador se coloca después de la variable, ésta se utiliza en la expresión y después se incrementa o decrementa en `1`.
- Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables a través de todas las plataformas que soportan Java.
- Java es un lenguaje fuertemente tipificado; requiere que todas las variables tengan un tipo.
- Las variables locales se declaran dentro de los métodos y no se les asignan valores predeterminados.
- A las variables declaradas fuera de los métodos se les asignan valores predeterminados. Las variables de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor `0` de manera predeterminada. Las variables de tipo `boolean` reciben el valor `false` de manera predeterminada.

TERMINOLOGÍA

`--`, operador
`?:`, operador
`++`, operador
acción
actividad
algoritmo
bloque
ciclo infinito
condición de continuación de ciclo
contador de ciclo
conversión implícita
cuerpo de un ciclo
decisión
división de enteros
`double`
ejecución secuencial
error de sintaxis
error lógico
estado de acción
estado final
estado inicial
estructura de control
estructura de control apilada
estructura de repetición
estructura de secuencia
estructura de selección
estructura de selección doble
estructura de selección simple
estructuras de control anidadas
estructuras de control de una sola entrada/una sola salida
expresión condicional
expresión de acción
expresión de creación de instancia de clase
flecha de transición
`if`, instrucción
`if...else`, instrucción
inicialización
instrucción de control
instrucción vacía (`;`)
Lenguaje de Modelado Unificado (UML)
mejoramiento de arriba a abajo, paso a paso
modelo acción/decisión
nota
operador condicional (`?:`)
operador de conversión de tipos, (*tipo*)
operador de decremento (`--`)
operador de incremento (`++`)
operador de postdecremento
operador de postincremento

http://librosysolucionarios.net

operador de predecremento
operador de preincremento
operador unario
operadores de asignación
  compuestos aritméticos: +=, -=, *=, /= y %=
programación estructurada
promoción
repetición
repetición controlada por centinela
repetición controlada por contador
repetición definida
repetición indefinida
seudocódigo
símbolo de estados de acción
símbolo de fusión
tipos primitivos
valor centinela
`while`, instrucción

EJERCICIOS DE AUTOEVALUACIÓN

4.1 Complete los siguientes enunciados:

a) Todos los programas pueden escribirse en términos de tres tipos de estructuras de control: _____, _____ y _____.

b) La instrucción _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra acción cuando esa condición es falsa.

c) Al proceso de repetir un conjunto de instrucciones un número específico de veces se le llama repetición _____.

d) Cuando no se sabe de antemano cuántas veces se repetirá un conjunto de instrucciones, se puede usar un valor _____ para terminar la repetición.

4.2 Escriba cuatro instrucciones distintas en Java, en donde cada una sume `1` a la variable entera `x`.

4.3 Escriba instrucciones en Java para realizar cada una de las siguientes tareas:

a) Asignar la suma de `x` e `y` a `z`, e incrementar `x` en `1` después del cálculo. Use sólo una instrucción.

b) Evaluar si la variable `cuenta` es mayor que `10`. De ser así, imprimir `"Cuenta es mayor que 10"`.

c) Decrementar la variable `x` en `1`, luego restarla a la variable `total`. Use sólo una instrucción.

d) Calcular el residuo después de dividir `q` entre `divisor`, y asignar el resultado a `q`. Escriba esta instrucción de dos maneras distintas.

4.4 Escriba una instrucción en Java para realizar cada una de las siguientes tareas:

a) Declarar las variables `suma` y `x` como de tipo `int`.

b) Asignar `1` a la variable `x`.

c) Asignar `0` a la variable `suma`.

d) Sumar la variable `x` a `suma` y asignar el resultado a la variable `suma`.

e) Imprimir la cadena `"La suma es: "`, seguida del valor de la variable `suma`.

4.5 Combine las instrucciones que escribió en el ejercicio 4.4 para formar una aplicación en Java que calcule e imprima la suma de los enteros del `1` al `10`. Use una instrucción `while` para iterar a través de las instrucciones de cálculo e incremento. El ciclo debe terminar cuando el valor de `x` se vuelva `11`.

4.6 Determine el valor de cada una de las siguientes variables, después de realizar el cálculo. Suponga que, cuando se empiece a ejecutar cada una de las instrucciones, todas las variables son de tipo `int` y tienen el valor `5`.

a) `producto *= x++;`

b) `cociente /= ++x;`

4.7 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código:

a)
```
while ( c <= 5 ) {
producto *= c;
++c;
```

b)
```
if ( genero == 1 )
   System.out.println( "Mujer" );
else;
   System.out.println( "Hombre" );
```

4.8 ¿Qué está mal en la siguiente instrucción `while`?

```
while ( z >= 0 )
   suma += z;
```

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

4.1 a) Secuencia, selección, repetición. b) `if...else`. c) Controlada por contador (o definida). d) Centinela, de señal, de prueba o de bandera.

http://librosysolucionarios.net

4.2
```
x = x + 1;
x += 1;
++x;
x++;
```

4.3
a)
```
z = x++ + y;
```
b)
```
if ( cuenta > 10 )
   System.out.println( "Cuenta es mayor que 10" );
```
c)
```
total -= --x;
```
d)
```
q %= divisor;
q = q % divisor;
```

4.4
a)
```
int suma, x;
```
b)
```
x = 1;
```
c)
```
suma = 0;
```
d)
```
suma += x; o suma = suma + x;
```
e)
```
System.out.println( "La suma es: " + suma );
```

4.5 El programa se muestra a continuación:

```
1  // Calcular la suma de los enteros del 1 al 10
2  public class Calcular {
3
4     public static void main( String args[] )
5     {
6        int suma, x;
7
8        x = 1;
9        suma = 0;
10
11       while ( x <= 10 ) {
12          suma += x;
13          ++x;
14       }
15
16       System.out.println( "La suma es: " + suma );
17
18    } // fin de main
19
20 } // fin de la clase Calcular
```

4.6 a) `producto = 25, x = 6`

b) `cociente = 0, x = 6`

4.7 a) Error: Falta la llave derecha de cierre del cuerpo de la instrucción `while`.
Corrección: Agregar una llave derecha de cierre después de la instrucción `++c;`.

b) Error: El punto y coma después de `else` produce un error lógico. La segunda instrucción de salida siempre se ejecutará.
Corrección: Quitar el punto y coma después de `else`.

4.8 El valor de la variable `z` nunca se cambia en la instrucción `while`. Por lo tanto, si la condición de continuación de ciclo (`z >= 0`) es verdadera, se crea un ciclo infinito. Para evitar que ocurra un ciclo infinito, `z` debe decrementarse de manera que eventualmente se vuelva menor que `0`.

EJERCICIOS

4.9 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código. [*Nota:* Puede haber más de un error en cada fragmento de código.]

```
a) if ( edad >= 65 );
      System.out.println( "Edad es mayor o igual que 65" );
   else
      System.out.println( "Edad es menor que 65 )";
b) int x = 1, total;
   while ( x <= 10 ) {
      total += x;
      ++x;
   }
c) while ( x <= 100 )
      total += x;
      ++x;
d) while ( y > 0 ) {
      System.out.println( y );
      ++y;
```

4.10 ¿Qué es lo que imprime el siguiente programa?

```
1  public class Misterio {
2
3     public static void main( String args[] )
4     {
5        int y, x = 1, total = 0;
6
7        while ( x <= 10 ) {
8           y = x * x;
9           System.out.println( y );
10          total += y;
11          ++x;
12       }
13
14       System.out.println( "El total es " +  total );
15
16    }  // fin de main
17
18 }  // fin de la clase Misterio
```

Para los ejercicios 4.11 a 4.14, realice cada uno de los siguientes pasos:

a) Lea el enunciado del problema.
b) Formule el algoritmo utilizando seudocódigo y el proceso de mejoramiento de arriba a abajo, paso a paso.
c) Escriba un programa en Java.
d) Pruebe, depure y ejecute el programa en Java.
e) Procese tres conjuntos completos de datos.

4.11 Los conductores se preocupan acerca del kilometraje de sus automóviles. Un conductor ha llevado el registro de varios reabastecimientos de gasolina, registrando los kilómetros conducidos y los litros usados en cada reabastecimiento. Desarrolle una aplicación en Java que reciba como entrada los kilómetros conducidos y los litros usados (ambos como enteros) por cada reabastecimiento. El programa debe calcular y mostrar los kilómetros por litro obtenidos en cada reabastecimiento, y debe imprimir el total de kilómetros por litro obtenidos en todos los reabastecimientos hasta este punto. Todos los cálculos del promedio deben producir resultados en números de punto flotante. Use cuadros de diálogo de entrada para obtener los datos del usuario.

4.12 Desarrolle una aplicación en Java que determine si el cliente de una tienda de departamentos se ha excedido del límite de crédito en una cuenta. Para cada cliente se tienen los siguientes datos:

a) el número de cuenta,
b) el saldo al inicio del mes,
c) el total de todos los artículos cargados por el cliente en el mes,
d) el total de todos los créditos aplicados a la cuenta del cliente en el mes, y
e) el límite de crédito permitido.

http://librosysolucionarios.net

El programa debe recibir como entrada cada uno de estos datos en forma de números enteros mediante cuadros de diálogo de entrada, debe calcular el nuevo saldo *(=saldo inicial + cargos – créditos)*, mostrar el nuevo balance y determinar si éste excede el límite de crédito del cliente. Para los clientes cuyo límite de crédito sea excedido, el programa debe mostrar el mensaje "Se excedió el límite de su crédito".

4.13 Una empresa grande paga a sus vendedores mediante comisiones. Los vendedores reciben $200 por semana, más el 9% de sus ventas brutas durante esa semana. Por ejemplo, un vendedor que vende $5000 de mercancía en una semana, recibe $200 más el 9% de 5000, o un total de $650. Usted acaba de recibir una lista de los artículos vendidos por cada vendedor. Los valores de estos artículos son los siguientes:

Artículo	Valor
1	239.99
2	129.75
3	99.95
4	350.89

Desarrolle una aplicación en Java que reciba como entrada los artículos vendidos por un vendedor durante la última semana, y que calcule y muestre los ingresos de ese vendedor. No hay límite en cuanto al número de artículos que un vendedor puede vender.

4.14 Desarrolle una aplicación en Java que determine el sueldo bruto para cada uno de tres empleados. La empresa paga la "cuota normal" en las primeras 40 horas de trabajo de cada empleado, y paga "cuota y media" en todas las horas trabajadas que excedan de 40. Usted recibe una lista de los empleados de la empresa, el número de horas que trabajó cada empleado la semana pasada y la tarifa por horas de cada empleado. Su programa debe recibir como entrada esta información para cada empleado, debe determinar y mostrar el sueldo bruto de cada empleado. Utilice cuadros de diálogo de entrada para introducir los datos.

4.15 El proceso de encontrar el valor más grande (es decir, el máximo de un grupo de valores) se utiliza frecuentemente en aplicaciones de computadora. Por ejemplo, un programa para determinar el ganador de un concurso de ventas recibe como entrada el número de unidades vendidas por cada vendedor. El vendedor que haya vendido más unidades es el que gana el concurso. Escriba un programa en seudocódigo y después una aplicación en Java que reciba como entrada una serie de 10 números de un solo dígito como caracteres, y que determine e imprima el mayor de los números. Su programa debe utilizar cuando menos las siguientes tres variables:

a) `contador`: Un contador para contar hasta 10 (es decir, para llevar el registro de cuántos números se han introducido, y para determinar cuando se hayan procesado los 10 números);
b) `numero`: El dígito actual introducido en el programa;
c) `mayor`: El número más grande encontrado hasta ahora.

4.16 Escriba una aplicación en Java que utilice ciclos para imprimir la siguiente tabla de valores:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

4.17 Utilizando una metodología similar a la del ejercicio 4.15, encuentre los dos valores más grandes de los 10 dígitos introducidos. [*Nota:* Puede introducir cada número sólo una vez.]

4.18 Modifique el programa de la figura 4.11 para validar sus entradas. Para cualquier entrada, si el valor introducido es distinto de 1 o 2, debe seguir iterando hasta que el usuario introduzca un valor correcto.

4.19 ¿Qué es lo que imprime el siguiente programa?

```
1  public class Misterio2 {
2
3     public static void main ( String args[] )
```

(Parte 1 de 2.)

```
4     {
5        int cuenta = 1;
6
7        while ( cuenta <= 10 ) {
8           System.out.println( cuenta % 2 == 1 ? "*****" : "++++++++" );
9
10          ++cuenta;
11       }
12
13    }  // fin de main
14
15 }  // fin de la clase Misterio2
```

(Parte 2 de 2.)

4.20 ¿Qué es lo que imprime el siguiente programa?

```
1  public class Misterio3 {
2
3     public static void main ( String args[] )
4     {
5        int fila = 10, columna;
6
7        while ( fila >= 1 ) {
8           columna = 1;
9
10          while ( columna <= 10 ) {
11             System.out.print( fila % 2 == 1 ? "<" : ">" );
12             ++columna;
13          }
14
15          --fila;
16          System.out.println();
17       }
18
19    }  // fin de main
20
21 }  // fin de la clase Misterio3
```

4.21 (*Problema del* `else` *suelto.*) Determine la salida de cada uno de los siguientes fragmentos de código, cuando `x` es `9` e `y` es `11`, y cuando `x` es `11` e `y` es `9`. Observe que el compilador ignora la sangría en un programa en Java. Además, el compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique de otra forma mediante la colocación de llaves ({}). A primera vista, el programador tal vez no esté seguro de cuál `if` corresponde a cuál `else`; esta situación se conoce como el "problema del `else` suelto". Hemos eliminado la sangría del siguiente código para hacer el problema más retador. [*Sugerencia:* Aplique las convenciones de sangría que ha aprendido.]

a)
```
if ( x < 10 )
   if ( y > 10 )
   System.out.println( "*****" );
   else
   System.out.println( "#####" );
   System.out.println( "$$$$$" );
```

b)
```
if ( x < 10 ) {
   if ( y > 10 )
   System.out.println( "*****" );
```

http://librosysolucionarios.net


```
   }
   else {
   System.out.println( "#####" );
   System.out.println( "$$$$$" );
   }
```

4.22 (*Otro problema de* `else` *suelto.*) Modifique el código dado para producir la salida que se muestra en cada parte del problema. Utilice las técnicas de sangría apropiadas. No haga modificaciones en el código, sólo inserte llaves o modifique la sangría del código. El compilador ignora la sangría en un programa en Java. Hemos eliminado la sangría en el código dado, para hacer el problema más retador. [*Nota:* Es posible que no se requieran modificaciones en algunas de las partes.]

```
if ( y == 8 )
if ( x == 5 )
System.out.println( "@@@@@" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
System.out.println( "&&&&&" );
```

a) Suponiendo que `x = 5` e `y = 8`, se produce la siguiente salida:
```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo que `x = 5` e `y = 8`, se produce la siguiente salida:
```
@@@@@
```

c) Suponiendo que `x = 5` e `y = 8`, se produce la siguiente salida:
```
@@@@@
&&&&&
```

d) Suponiendo que `x = 5` e `y = 7`, se produce la siguiente salida. [*Nota:* Las tres últimas instrucciones de salida después del `else` forman parte de un bloque.]
```
#####
$$$$$
&&&&&
```

4.23 Escriba un applet (subprograma) que lea el tamaño del lado de un cuadrado y que muestre un cuadrado hueco de ese tamaño, compuesto de asteriscos, mediante el uso del método `drawString` dentro del método `paint` de su applet. Use un cuadro de diálogo de entrada para recibir el tamaño como entrada por parte del usuario. Su programa debe funcionar con cuadrados que tengan lados de todas las longitudes entre 1 y 20.

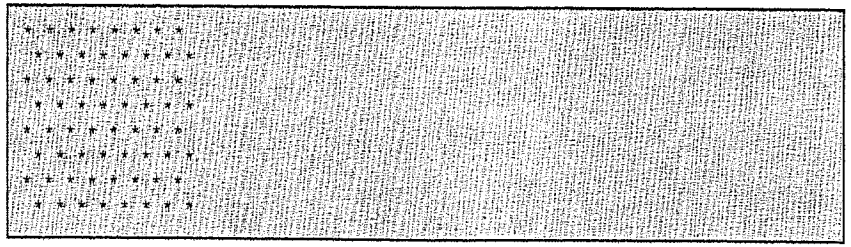
4.24 Un palíndromo es una secuencia de caracteres que se lee igual al derecho y al revés. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos es un palíndromo: 12321, 55555, 45554 y 11611. Escriba una aplicación que lea un entero de cinco dígitos y determine si es un palíndromo. Si el número no es de cinco dígitos, mostrar un cuadro de diálogo con un mensaje de error indicando el problema al usuario. Cuando el usuario desaparezca el mensaje de error, se le debe permitir introducir un nuevo valor.

4.25 Escriba una aplicación que reciba como entrada un entero que contenga sólo 0s y 1s (es decir, un entero "binario"), y que imprima su equivalente decimal. [*Sugerencia:* Use los operadores residuo y división para elegir los dígitos del "número binario" uno a la vez, de derecha a izquierda. En el sistema numérico decimal, el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1000, etc. El número decimal 234 puede interpretarse como 4 * 1 + 3 * 10 + 2 * 100. En el sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etc. El equivalente decimal del número binario 1101 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, o 1 + 0 + 4 + 8, o 13.]

4.26 Escriba una aplicación que utilice sólo las instrucciones de salida

```
System.out.print( "* " );
System.out.print( " " );
System.out.println();
```

para mostrar el patrón de tablero de damas que se muestra a continuación. Observe que una llamada al método `System.out.println` sin argumentos hace que el programa imprima un solo carácter de nueva línea. [*Sugerencia:* Se requieren estructuras de repetición.]



4.27 Escriba una aplicación que muestre en la ventana de comandos los múltiplos del entero 2 (es decir, 2, 4, 8, 16, 32, 64, etc.). Su ciclo no debe terminar (es decir, debe crear un ciclo infinito). ¿Qué ocurre cuando ejecuta este programa?

4.28 ¿Qué está mal en la siguiente instrucción? Proporcione la instrucción correcta para agregar uno a la suma de `x` e `y`.

```
System.out.println( ++( x + y ) );
```

4.29 Escriba una aplicación que lea tres valores distintos de cero, introducidos por el usuario mediante cuadros de diálogo de entrada, y que determine si pueden representar los lados del triángulo, imprimiendo un mensaje si es así.

4.30 Escriba una aplicación que lea tres enteros distintos de cero y determine si pueden representar los lados de un triángulo recto.

4.31 Una compañía desea transmitir datos a través del teléfono, pero le preocupa que sus teléfonos puedan estar intervenidos. Le ha pedido a usted que escriba un programa que cifre sus datos, de manera que éstos puedan transmitirse con más seguridad. Todos sus datos se transmiten como enteros de cuatro dígitos. Su aplicación debe leer un entero de cuatro dígitos introducido por el usuario en un cuadro de diálogo de entrada, y debe cifrar ese número de la siguiente manera: reemplace cada dígito con el resultado de sumar 7 al dígito y obtener el residuo después de dividir el nuevo valor entre 10. Luego intercambie el primer dígito con el tercero, e intercambie el segundo dígito con el cuarto. Después imprima el entero cifrado. Escriba una aplicación separada que reciba como entrada un entero de cuatro dígitos cifrado, y que lo descifre para formar el número original.

4.32 El factorial de un entero n no negativo se escribe como $n!$ y se define de la siguiente manera:
$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (para valores de n mayores o iguales a 1)

y

$n! = 1$ (para $n = 0$)

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es 120.

a) Escriba una aplicación que lea un entero no negativo de un cuadro de diálogo de entrada y que calcule e imprima su factorial.

b) Escriba una aplicación que estime el valor de la constante matemática e, utilizando la fórmula

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Escriba una aplicación que calcule el valor de e^x, utilizando la fórmula

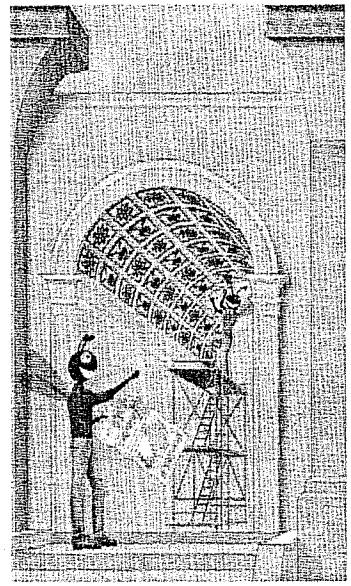
$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

5

Instrucciones de control: Parte 2

Objetivos

- Utilizar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones de manera repetitiva en un programa.
- Comprender la selección múltiple utilizando la instrucción de selección `switch`.
- Utilizar las instrucciones de control de programa `break` y `continue`.
- Utilizar los operadores lógicos.



¿Quién puede controlar su destino?
William Shakespeare

La llave usada siempre es brillante.
Benjamín Franklin

El hombre es un animal que fabrica herramientas.
Benjamín Franklin

La inteligencia... es la facultad de hacer objetos artificiales, especialmente herramientas para hacer herramientas.
Henri Bergson

http://librosysolucionarios.net

Plan general

5.1 Introducción
5.2 Fundamentos de la repetición controlada por contador
5.3 Instrucción de repetición `for`
5.4 Ejemplos sobre el uso de la instrucción `for`
5.5 Instrucción de repetición `do...while`
5.6 Instrucción de selección múltiple `switch`
5.7 Instrucciones `break` y `continue`
5.8 Instrucciones `break` y `continue` etiquetadas
5.9 Operadores lógicos
5.10 Resumen sobre programación estructurada
5.11 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar los estados y actividades de los objetos

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

5.1 Introducción

El capítulo 4 nos introdujo a los tipos de bloques de construcción disponibles para solucionar problemas; en dichos bloques de construcción empleamos los principios, ya comprobados, de la construcción de programas. En este capítulo continuaremos nuestra presentación de la teoría y los principios de la programación estructurada, ya que explicaremos el resto de las instrucciones de control en Java. Al igual que en el capítulo 4, las técnicas de Java que usted aprenda aquí se aplican a la mayoría de los lenguajes de alto nivel. Cuando, en el capítulo 8, comencemos nuestro análisis formal de la programación orientada a objetos en Java, veremos que las instrucciones de control que estudiaremos en este capítulo y las que vimos en el capítulo 4 son útiles para crear y manipular objetos.

5.2 Fundamentos de la repetición controlada por contador

Esta sección utiliza la instrucción de repetición **`while`**, presentada en el capítulo 4, para formalizar los elementos requeridos para llevar a cabo la repetición controlada por contador. Este tipo de repetición requiere

1. una *variable de control* (o contador de ciclo),
2. el *valor inicial* de la variable de control,
3. la cantidad de *incremento* (o *decremento*) con la que se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como *cada iteración del ciclo*) y
4. la *condición* que evalúa el *valor final* de la variable de control (es decir, si el ciclo debe continuar o no).

Para ver estos elementos de la repetición controlada por contador, considere el applet (subprograma) que se muestra en la figura 5.1, el cual dibuja 10 líneas desde su método **`paint`**. Recuerde que en el capítulo 3 vimos que para poder cargar un applet en un contenedor de applets se requiere de un documento HTML. El documento HTML que acompaña a este applet (en CD) contiene un elemento **`applet`** con una anchura de **`275`** píxeles y una altura de **`110`** píxeles.[1]

Recuerde que en el capítulo 3 también vimos que el contenedor de applets carga la clase del applet, crea un objeto de la clase y llama a los métodos **`init`**, **`start`** y **`paint`** de ese objeto. En **`paint`** (líneas 10 a 22),

1. No mostraremos el documento HTML de cada applet en lo que resta del libro, a menos que el documento presente una nueva característica. Sin embargo, en el CD que acompaña a este libro proporcionamos un documento HTML de cada applet para que usted pueda probarlo en un contenedor de applets.

http://librosysolucionarios.net


```
1  // Fig. 5.1: ContadorWhile.java
2  // Repetición controlada por contador.
3  import java.awt.Graphics;
4
5  import javax.swing.JApplet;
6
7  public class ContadorWhile extends JApplet {
8
9     // dibujar líneas en el fondo del applet
10    public void paint( Graphics g )
11    {
12       super.paint( g );  // llamar al método paint heredado de JApplet
13
14       int contador = 1;  // inicialización
15
16       while ( contador <= 10 ) {  // condición de repetición
17          g.drawLine( 10, 10, 250, contador * 10 );
18          ++contador;  // incremento
19
20       } // fin de instrucción while
21
22    } // fin del método paint
23
24 } // fin de la clase ContadorWhile
```

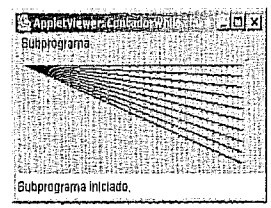


Figura 5.1 Repetición controlada por contador con la instrucción de repetición **`while`**.

los elementos de la repetición controlada por contador se definen en las líneas 14 a 20. La línea 14 declara la *variable de control* (**`contador`**) como un entero, reserva espacio para esta variable en memoria y establece su *valor inicial* en **`1`**. Recuerde que en la sección 4.10 vimos que la declaración de una variable puede especificar el valor inicial de esa variable. La declaración e inicialización de **`contador`** también pudo haberse logrado con las siguientes instrucciones de declaración y asignación de variables locales:

```
int contador; // declarar contador
contador = 1; / asignar 1 a contador
```

Nosotros utilizaremos ambas técnicas para inicializar variables en este libro.

La instrucción **`while`**, en línea 17, utiliza la referencia **`g`** de la clase **`Graphics`**, la cual hace referencia al objeto **`Graphics`** del applet, para enviar el mensaje **`drawLine`** al objeto **`Graphics`**, pidiéndole que dibuje una línea. Recuerde que "enviar un mensaje a un objeto" en realidad significa llamar a un método para que realice una tarea. Por lo tanto, podemos decir que la línea 17 utiliza a **`g`** para llamar al método **`drawLine`** del objeto **`Graphics`**. El método **`drawLine`** requiere cuatro argumentos, los cuales representan la primera coordenada *x*, la primera coordenada *y*, la segunda coordenada *x* y la segunda coordenada *y* de la línea. En este ejemplo, la segunda coordenada *y* cambia de valor durante cada iteración en el ciclo, con el cálculo **`contador * 10`**. Ese cambio hace que el segundo punto (el punto final de la línea) se mueva en cada llamada a **`drawLine`** 10 píxeles hacia abajo del área visible del applet.

En la línea 18, la instrucción `while` *incrementa* la variable de control en 1 para cada iteración del ciclo. La *condición* de continuación de ciclo en `while` evalúa si el valor de la variable de control es menor o igual a `10` (el *valor final* para el que la condición es `true`). Observe que el programa ejecuta el cuerpo de su instrucción `while`, aun cuando la variable de control sea `10`. El ciclo termina cuando la variable de control se pasa de `10` (es decir, cuando `contador` se vuelve `11`).



Error común de programación 5.1

Como los valores de punto flotante pueden ser aproximados, controlar ciclos con variables de punto flotante puede producir valores incorrectos en el contador, y pruebas imprecisas para la terminación del ciclo. Por esta razón, los programas deben controlar los ciclos de contadores con enteros.



Buena práctica de programación 5.1

Coloque líneas en blanco por encima y debajo de las instrucciones de control, y aplique sangría a los cuerpos de las instrucciones de control que estén dentro de los encabezados de éstas (primeras líneas), para dar a los programas una apariencia bidimensional que mejore la legibilidad.

El programa de la figura 5.1 puede hacerse más conciso si se inicializa `contador` en 0 y se preincrementa `contador` en la condición `while` de la siguiente forma:

```
while ( ++contador <= 10 )    // condición de repetición
 g.drawLine( 10, 10, 250, contador * 10 );
```

Este código ahorra una instrucción (y elimina la necesidad de usar llaves alrededor del cuerpo del ciclo), ya que la condición de `while` realiza el incremento antes de evaluar la condición. (Recuerde que en la sección 4.12 vimos que la precedencia de `++` es mayor que la de `<=`.) La codificación en esta forma tan condensada requiere de práctica y hace que el código sea más difícil de leer, depurar, modificar y mantener. En general, es mejor evitar codificar de esta manera. Sin embargo, en ocasiones encontrará código como éste, por lo que debe comprender la manera en que funciona.

5.3 Instrucción de repetición `for`

La sección 5.2 presentó los cuatro componentes requeridos para la repetición controlada por contador. La instrucción `while` puede usarse para implementar cualquier ciclo controlado por contador. Sin embargo, Java también cuenta con la instrucción de repetición `for`, la cual especifica los detalles sobre la repetición controlada por contador en una sola línea de código. Para ilustrar el poder de la instrucción `for`, implementaremos nuevamente el applet de la figura 5.1. El nuevo applet aparece en la figura 5.2. El documento HTML para probar este applet define una anchura de `275` píxeles y una altura de `110` píxeles.

El método `paint` del applet opera de la siguiente manera: cuando la instrucción `for` (líneas 16 y 17) comienza a ejecutarse, la variable de control `contador` se declara y se inicializa en `1`. (Recuerde que en la sección 5.2 vimos que los primeros dos elementos de la repetición controlada por contador son la *variable de control* y su *valor inicial*.) A continuación, el programa comprueba la *condición* de continuación de ciclo, `contador <= 10`. La condición contiene el *valor final* (`10`) de la variable de control. Como el valor inicial de `contador` es `1`, la condición inicialmente es verdadera. Por lo tanto, la instrucción del cuerpo (línea 17) dibuja una línea. Después de ejecutar el cuerpo del ciclo, el programa *incrementa* la variable `contador` en la expresión `contador++`. Luego, el programa realiza la prueba de control de ciclo nuevamente, para determinar si el programa debe continuar con la siguiente iteración del ciclo. En este punto el valor de la variable de control es 2, por lo que la condición es verdadera (no pasa del valor final); por lo tanto, el programa ejecuta la instrucción del cuerpo otra vez (es decir, la siguiente iteración del ciclo). Este proceso continúa hasta que el valor de `contador` se incrementa a `11`, esto ocasionará que la condición de continuidad del ciclo falle y que la repetición termine. Luego, el programa ejecuta la primera instrucción después del `for`. (En este caso, el método `paint` termina, ya que el programa llega al final de `paint`.)

Observe que la figura 5.2 utiliza la condición de continuación de ciclo `contador <= 10`. Si el programador especificara incorrectamente `contador < 10` como la condición, el ciclo se ejecutaría sólo nueve veces. Éste es un error lógico, que se conoce como *error por desplazamiento en uno*.



Error común de programación 5.2

Usar un operador relacional incorrecto o usar un valor final incorrecto en un contador de ciclo en la condición de una instrucción de repetición, puede provocar un error por desplazamiento en uno.

http://librosysolucionarios.net


```
1  // Fig. 5.2: ContadorFor.java
2  // Repetición controlada por contador con la instrucción for.
3  import java.awt.Graphics;
4
5  import javax.swing.JApplet;
6
7  public class ContadorFor extends JApplet {
8
9     // dibujar líneas en el fondo del applet
10    public void paint( Graphics g )
11    {
12       super.paint( g );  // llamar al método paint heredado de JApplet
13
14       // el encabezado de la instrucción for incluye la inicialización,
15       // la condición de repetición y el incremento
16       for ( int contador = 1; contador <= 10; contador++ )
17          g.drawLine( 10, 10, 250, contador * 10 );
18
19    } // fin del método paint
20
21 } // fin de la clase ContadorFor
```

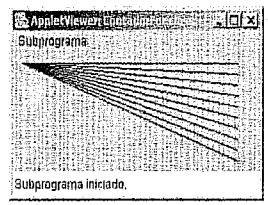


Figura 5.2 Repetición controlada por contador con la instrucción de repetición `for`.



Buena práctica de programación 5.2

Usar el valor final en la condición de una instrucción `while` o `for` y usar el operador relacional `<=`, son estrategias que ayudan a evitar los errores por desplazamiento en uno. Para un ciclo que imprime los valores 1 a 10, la condición de continuación de ciclo debe ser `contador <= 10` en vez de `contador < 10` (que provoca un error por desplazamiento en uno) o `contador < 11` (lo cual es correcto). Muchos programadores prefieren lo que se conoce como conteo de base cero, en el que para contar 10 veces, `contador` se debe inicializar en cero y la prueba de continuación de ciclo debe ser `contador < 10`.

La figura 5.3 analiza con más detalle la instrucción `for` de la figura 5.2. A la primera línea del `for` (incluyendo la palabra clave `for` y todo lo que hay en paréntesis después de ella), línea 16 en la figura 5.2, se le conoce como *encabezado de la instrucción `for`*, o simplemente *encabezado del `for`*. Observe que la instrucción `for` "lo hace todo": especifica cada uno de los elementos necesarios para la repetición controlada por contador mediante una variable de control. Si hay más de una instrucción en el cuerpo del `for`, se requieren llaves ({ y }) para definir el cuerpo del ciclo.

El formato general de la instrucción `for` es

```
for ( inicialización; condiciónDeContinuaciónDeCiclo; incremento )
    instrucción
```

en donde la expresión *inicialización* nombra a la variable de control del ciclo y proporciona su valor inicial, *condiciónDeContinuaciónDeCiclo* es la condición que determina si el ciclo debe continuar ejecutándose (esta condición contiene el valor final para el cual la condición es verdadera), e *incremento* modifica el valor de la variable de control, de manera que la condición de continuación de ciclo eventualmente se vuelva falsa. En la

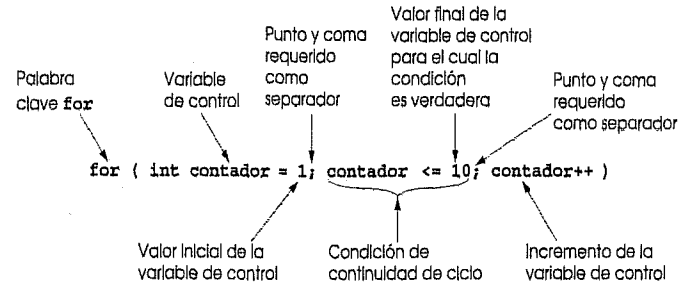


Figura 5.3 Componentes del encabezado de la instrucción `for`.

mayoría de los casos, la instrucción `for` puede representarse con una instrucción `while` equivalente,[2] de la siguiente manera:

```
inicialización;
while ( condiciónDeContinuaciónDeCiclo ) {
instrucción
incremento;
}
```

Generalmente, las instrucciones `for` se utilizan para la repetición controlada por contador, y las instrucciones `while` se utilizan para la repetición controlada por centinela. Sin embargo, `while` y `for` pueden usarse para cualquier tipo de repetición.

Si la expresión de *inicialización* en el encabezado de `for` declara a la variable de control (es decir, que el tipo de la variable de control se especifique antes del nombre de la variable, como en la figura 5.3), esta variable puede usarse solamente en esa instrucción `for`; por lo tanto, no existirá fuera de la instrucción `for`. Este uso restringido del nombre de la variable de control se conoce como *alcance*. El alcance de una variable define en dónde puede utilizarse dentro de un programa. Por ejemplo, una variable local puede usarse solamente en el método que la declare. El alcance se describe con detalle en el capítulo 6, Métodos.

Error común de programación 5.3

Se producirá un error de sintaxis si la variable de control de una instrucción `for` se declara en la sección de inicialización del encabezado del `for` y se utiliza después del cuerpo del `for`.

Como veremos en la sección 5.4, las expresiones de *inicialización* e *incremento* pueden ser listas de expresiones separadas por comas, las cuales permiten al programador usar varias expresiones de inicialización o varias expresiones de incremento. Por ejemplo, puede haber varias variables de control en un solo `for` que deban inicializarse e incrementarse.

Buena práctica de programación 5.3

Coloque sólo las expresiones involucradas con las variables de control en las secciones de inicialización e incremento de una instrucción `for`. Las manipulaciones de otras variables deben aparecer ya sea antes del ciclo (si se ejecutan sólo una vez, como las instrucciones de inicialización) o en el cuerpo del mismo (si se ejecutan una vez por cada iteración del ciclo, como las instrucciones de incremento o decremento).

Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java supone que la condición de continuación de ciclo es verdadera, lo que en consecuencia produce un ciclo infinito. Podría omitirse la expresión de *inicialización* si el programa inicializa la variable de control antes del ciclo. Podría omitirse la expresión de *incremento* si el programa lo calcula con instrucciones en el cuerpo del

2. En la sección 5.7 mostraremos un caso en el que una instrucción `for` no puede representarse con una instrucción `while` equivalente.

ciclo o si no se necesita un incremento. La expresión de incremento en un `for` actúa como una instrucción individual al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son expresiones de incremento equivalentes en una instrucción `for`. Muchos programadores prefieren usar `contador++`, ya que un ciclo `for` evalúa su expresión de incremento después de que se ejecuta su cuerpo. Por lo tanto, la forma de postincremento parece ser más natural. En este caso, la variable que va a incrementarse no aparece en una expresión más grande, por lo que tanto el preincremento como el postdecremento tienen el mismo efecto. Los dos puntos y coma en el encabezado `for` son obligatorios.

Error común de programación 5.4

Usar comas en vez de los dos puntos y coma requeridos en un encabezado `for` es un error de sintaxis.

Error común de programación 5.5

Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho de un encabezado `for`, hace que el cuerpo de ese `for` sea una instrucción vacía. Esto generalmente es un error lógico.

Error común de programación 5.6

Los ciclos infinitos ocurren cuando la condición de continuación de ciclo en una instrucción de repetición nunca se vuelve `false`. Para prevenir esta situación, asegúrese de que no haya un punto y coma inmediatamente después del encabezado de un `while` o `for`. En un ciclo controlado por contador, asegúrese de que la variable de control se incremente (o decremente) durante cada iteración del ciclo. En un ciclo controlado por centinela, asegúrese de que el valor centinela se introduzca en algún momento.

Las porciones de inicialización, condición de continuación de ciclo e incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`. Si `x` y `y` no se modifican en el cuerpo del ciclo, la instrucción

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( int j = 2; j <= 80; j += 5 )
```

El incremento de una instrucción `for` también puede ser negativo, en cuyo caso es en realidad un decremento, y el ciclo avanza en forma inversa.

Si la condición de continuación de ciclo es inicialmente `false`, el programa no ejecuta el cuerpo de la instrucción `for`. En vez de ello, la ejecución prosigue con la instrucción que va después del `for`.

Los programas frecuentemente muestran el valor de la variable de control o lo utilizan en cálculos en el cuerpo del ciclo. Sin embargo, este uso no es requerido. Es común utilizar la variable de control para controlar la repetición sin mencionarla siquiera en el cuerpo del `for`.

Tip para prevenir errores 5.1

Aunque el valor de la variable de control puede cambiarse en el cuerpo de un ciclo `for`, evite hacerlo ya que esta práctica puede ocasionar pequeños errores.

El diagrama de actividad de la instrucción `for` es similar al de la instrucción `while` (figura 4.5). La figura 5.4 muestra el diagrama de actividad de la instrucción `for` de la figura 5.2. El diagrama deja en claro que la inicialización se ejecuta primero antes de evaluar la prueba de continuación de ciclo por primera vez, y que el incremento ocurre *después* de que se ejecuta la instrucción del cuerpo.

5.4 Ejemplos sobre el uso de la instrucción `for`

Los siguientes ejemplos muestran técnicas para modificar la variable de control en una instrucción `for`. En cada caso, escribimos el encabezado `for` apropiado. Observe el cambio en el operador relacional para los ciclos que decrementan la variable de control.

http://librosysolucionarios.net

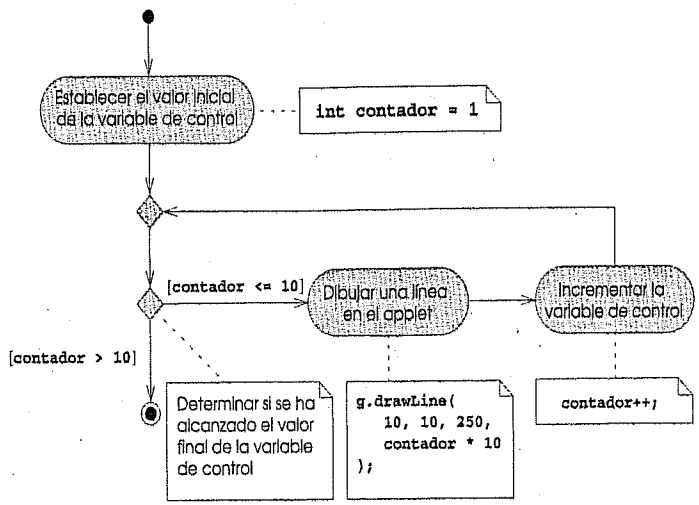


Figura 5.4 Diagrama de actividad de la instrucción `for`.

a) Modificar la variable de control de `1` a `100` en incrementos de `1`.

```
for ( int i = 1; i <= 100; i++ )
```

b) Modificar la variable de control de `100` a `1` en incrementos de `-1` (es decir, decrementos de `1`).

```
for ( int i = 100; i >= 1; i-- )
```

c) Modificar la variable de control de `7` a `77` en incrementos de `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Modificar la variable de control de `20` a `2` en incrementos de `-2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Modificar la variable de control con la siguiente secuencia de valores: `2, 5, 8, 11, 14, 17, 20`.

```
for ( int j = 2; j <= 20; j += 3 )
```

f) Modificar la variable de control con la siguiente secuencia de valores: `99, 88, 77, 66, 55, 44, 33, 22, 11, 0`.

```
for ( int j = 99; j >= 0; j -= 11 )
```



Error común de programación 5.7

No utilizar el operador relacional apropiado en la condición de continuación de un ciclo que cuente en forma regresiva (como usar `i <= 1` en un ciclo que cuente en forma regresiva hasta llegar a `1`) es generalmente un error lógico, que producirá resultados incorrectos cuando el programa se ejecute.

Sumar los enteros pares del 2 al 100

Ahora consideremos dos programas de ejemplo que demuestran usos simples de la instrucción `for`. La aplicación de la figura 5.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 100 y guardar el resultado en una variable `int` llamada `total`. Este programa es una aplicación, por lo que debe usarse el intérprete `java` para ejecutar la aplicación desde la ventana de comandos.

http://librosysolucionarios.net


```
1  // Fig. 5.5: Suma.java
2  // Suma de enteros con la instrucción for.
3  import javax.swing.JOptionPane;
4
5  public class Suma {
6
7     public static void main( String args[] )
8     {
9        int total = 0; // inicializar el total
10
11       // sumar los enteros pares del 2 al 100
12       for ( int numero = 2; numero <= 100; numero += 2 )
13          total += numero;
14
15       // mostrar el resultado
16       JOptionPane.showMessageDialog( null, "La suma es " + total,
17          "Suma de los enteros pares del 2 al 100",
18          JOptionPane.INFORMATION_MESSAGE );
19
20       System.exit( 0 );   // terminar la aplicación
21
22    } // fin de main
23
24 } // fin de la clase Suma
```



Figura 5.5 Sumatoria con la instrucción `for`.

El cuerpo de la instrucción `for` en las líneas 12 y 13 de la figura 5.5 podría fusionarse utilizando una coma de la siguiente manera:

```
for ( int numero = 2; numero <= 100; total += numero, numero += 2 )
   ; // instrucción vacía
```

De manera similar, la inicialización `total = 0` podría fusionarse en la sección de inicialización de la instrucción `for`.



Buena práctica de programación 5.4

Limite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

Cálculo del interés compuesto

El siguiente ejemplo utiliza la instrucción `for` para calcular el interés compuesto. Considere el siguiente problema:

Una persona invierte $1000.00 en una cuenta de ahorro que produce el 5% de interés. Suponiendo que todo el interés se deposita en la cuenta, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Use la siguiente fórmula para determinar los montos:

$$c = p(1 + r)^n$$

en donde

p es el monto que se invirtió originalmente (es decir, el monto principal)
r es la tasa de interés anual
n es el número de años
c es la cantidad depositada al final del *n*-ésimo año.

Este problema implica el uso de un ciclo que realiza los cálculos indicados para cada uno de los 10 años que el dinero permanece depositado. La solución es la aplicación que se muestra en la figura 5.6.

Las líneas 13 a 15 en el método **main** declaran las variables **cantidad**, **principal** y **tasa** de tipo **double**, e inicializan **principal** en **1000.0** y **tasa** en **0.05**. Java trata a las constantes de punto flotante, como **1000.0** y **0.05**, como de tipo **double**. De manera similar, Java trata a las constantes de números enteros, como 7 y **-22**, como de tipo **int**. Las líneas 18 y 19 declaran la referencia **formatoMoneda** de tipo

```
1  // Fig. 5.6: Interes.java
2  // Cálculo del interés compuesto.
3  import java.text.NumberFormat;  // clase para el formato numérico
4  import java.util.Locale;  // clase para información específica de cada país
5
6  import javax.swing.JOptionPane;
7  import javax.swing.JTextArea;
8
9  public class Interes {
10
11    public static void main( String args[] )
12    {
13       double cantidad;         // cantidad depositada al final de cada año
14       double principal = 1000.0;   // cantidad inicial sin intereses
15       double tasa = 0.05;          // tasa de interés
16
17       // crear NumberFormat para la moneda en dólares
18       NumberFormat formatoMoneda =
19          NumberFormat.getCurrencyInstance( Locale.US );
20
21       // crear JTextArea para mostrar la salida
22       JTextArea areaTextoSalida = new JTextArea();
23
24       // establecer la primera línea de texto en areaTextoSalida
25       areaTextoSalida.setText( "Año\tCantidad en depósito\n" );
26
27       // calcular la cantidad en depósito para cada uno de los diez años
28       for ( int anio = 1; anio <= 10; anio++ ) {
29
30          // calcular la nueva cantidad para el año especificado
31          cantidad = principal * Math.pow( 1.0 + tasa, anio );
32
33          // anexar una línea de texto a areaTextoSalida
34          areaTextoSalida.append( anio + "\t" +
35             formatoMoneda.format( cantidad ) + "\n" );
36
37       } // fin de instrucción for
38
39       // mostrar resultados
40       JOptionPane.showMessageDialog( null, areaTextoSalida,
41          "Interés compuesto", JOptionPane.INFORMATION_MESSAGE );
42
43       System.exit( 0 );  // terminar la aplicación
44
45    } // fin de main
46
47 } // fin de la clase Interes
```

Figura 5.6 Cálculo del interés compuesto con la instrucción **for**. (Parte 1 de 2.)

http://librosysolucionarios.net

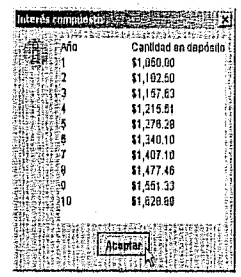


Figura 5.6 Cálculo del interés compuesto con la instrucción **for**. (Parte 2 de 2.)

NumberFormat, y la inicializan llamando al método estático ***getCurrencyInstance*** de **NumberFormat**. Este método devuelve un objeto **NumberFormat** que puede dar formato a los valores numéricos como cantidades monetarias (por ejemplo, en los Estados Unidos a los valores monetarios generalmente se les antepone un signo de dólares, $). El argumento para el método (**Locale.US**) indica que los valores monetarios deben mostrarse como dólares estadounidenses: deben comenzar con un signo de dólares ($), usar un punto decimal para separar los dólares de los centavos y utilizar una coma para indicar los miles (por ejemplo, $1,234.56). La clase **Locale** proporciona constantes que pueden utilizarse para personalizar este programa y representar valores monetarios de otros países, de manera que los formatos de moneda se muestren apropiadamente para cada *configuración regional* (es decir, el formato de moneda local de cada país). La clase **NumberFormat** (que se importó en la línea 3) se encuentra en el paquete **java.text**, y la clase **Locale** (que se importó en la línea 4) se encuentra en el paquete ***java.util***.

La línea 22 declara la referencia **areaTextoSalida** de tipo **JTextArea** y la inicializa con un nuevo objeto de la clase **JTextArea** (del paquete **javax.swing**). Un objeto **JTextArea** es un componente de GUI que puede mostrar muchas líneas de texto. El cuadro de diálogo de mensaje que muestra el objeto **JTextArea** determina la anchura y altura de este objeto, con base en la cadena que contiene. Presentamos este componente de GUI ahora, porque veremos muchos ejemplos a lo largo de este texto, en los cuales la salida de los programas contiene demasiadas líneas a mostrar en la pantalla. Como veremos posteriormente en este capítulo, un objeto **JTextArea** (cuando se combina con otro componente llamado **JScrollPane**) nos permite desplazarnos a través de las líneas de texto, de manera que podamos ver toda la salida del programa. Los métodos para colocar texto en un objeto **JTextArea** incluyen a ***setText*** y ***append***.

La línea 25 utiliza el método **setText** de **JTextArea** para colocar una cadena en el objeto **JTextArea** al que **areaTextoSalida** hace referencia. Inicialmente, un objeto **JTextArea** contiene una cadena vacía (es decir, una cadena que no contiene caracteres). La instrucción anterior reemplaza la cadena vacía con una que contiene los encabezados para nuestras dos columnas de salida: **"Año"** y **"Cantidad en depósito"**. Los encabezados de las columnas se separan con un carácter de tabulación (secuencia de escape **\t**). Además, la cadena contiene el carácter de nueva línea (secuencia de escape **\n**), el cual indica que cualquier texto que se agregue al objeto **JTextArea** empezará en la siguiente línea.

La instrucción **for** (líneas 28 a 37) ejecuta su cuerpo 10 veces, modificando la variable de control **anio** desde 1 hasta 10, en incrementos de 1. Este ciclo termina cuando la variable de control **anio** se vuelve **11**. (Observe que **anio** representa a n en el enunciado del problema.) Java no incluye un operador de exponenciación. Por lo tanto, utilizamos el método estático **pow** de la clase **Math** para este fin. **Math.pow(x,y)** calcula el valor de **x** elevado a la **y**-ésima potencia. El método **pow** acepta dos argumentos de tipo **double** y devuelve un valor **double**. La línea 31 realiza el cálculo del enunciado del problema,

$$c = p(1 + t)^n$$

en donde c es **cantidad**, p es **principal**, t es **tasa** y n es **anio**.

Las líneas 34 y 35 agregan, mediante **append**, más texto al final de **salidaAreaTexto**. El texto incluye el valor actual de **anio**, un carácter de tabulación (para posicionarse en la segunda columna), el resultado de la llamada al método **moneyFormat.format(cantidad)** (el cual da formato a la **cantidad**

como valor monetario estadounidense) y un carácter de nueva línea (para posicionar el cursor en el objeto `JTextArea` al principio de la siguiente línea).

Las líneas 40 y 41 muestran los resultados de los cálculos en un cuadro de diálogo de mensaje. Hasta ahora, el mensaje mostrado ha sido siempre una cadena. En este ejemplo, el segundo argumento es `salida AreaTexto`, un componente de GUI. Una característica interesante de la clase `JOptionPane` es que el mensaje que muestra con `showMessageDialog` puede ser una cadena o un componente de GUI, como `JTextArea`. En este ejemplo, el cuadro de diálogo de mensaje ajusta su propio tamaño para alojar el objeto `JTextArea`.

En este ejemplo declaramos las variables `cantidad`, `principal` y `tasa` como de tipo `double`. Tenemos que tratar con las partes fraccionarias de los dólares y, por lo tanto, necesitamos un tipo que permita puntos decimales en sus valores. Desafortunadamente, esta configuración puede ocasionar problemas. He aquí una simple explicación de lo que puede salir mal al utilizar `double` (o `float`) para representar cantidades en dólares (suponiendo que estas cantidades se muestran con dos dígitos a la derecha del punto decimal): Dos cantidades `double` en dólares almacenadas en el equipo podrían ser 14.234 (que normalmente se redondearía a 14.23 para mostrarse en pantalla) y 18.673 (que normalmente se redondearía a 18.67 para mostrarse en pantalla). Cuando se suman estas cantidades, producen el resultado interno 32.907, que normalmente se redondearía a 32.91 para mostrarse en pantalla. Por lo tanto, la salida podría aparecer así:

```
  14.23
+ 18.67
--------
  32.91
```

pero una persona que sume los números individuales, según se muestran, esperaría que la suma fuera 32.90.



Buena práctica de programación 5.5

No utilice variables de tipo `double` o `float` para realizar cálculos monetarios precisos. La imprecisión de los números de punto flotante puede causar errores que generen valores monetarios incorrectos. En estos ejercicios exploramos el uso de enteros para realizar los cálculos monetarios. (Nota: Algunos distribuidores independientes tienen a la venta bibliotecas de clases que realizan cálculos monetarios con precisión. Además, la API de Java cuenta con la clase `java.math.BigDecimal` para realizar cálculos con valores de punto flotante con precisión arbitraria.)

Observe que el cuerpo de la instrucción `for` contiene el cálculo `1.0 + tasa`, el cual aparece como argumento para el método `Math.pow`. De hecho, este cálculo produce el mismo resultado cada vez que se ejecuta en el ciclo, por lo que repetir el cálculo en cada iteración del ciclo es un proceso que desperdicia recursos.



Tip de rendimiento 5.1

En los ciclos, evite los cálculos para los cuales el resultado nunca cambia. (Nota: Muchos de los sofisticados compiladores optimizadores de la actualidad colocan dichos cálculos fuera de los ciclos en el código compilado.)

5.5 Instrucción de repetición `do...while`

La instrucción `do...while` es similar a la instrucción `while`; ya que el programa evalúa la condición de continuación de ciclo al principio, antes de ejecutar el cuerpo del ciclo. Si la condición es falsa, el cuerpo del ciclo `while` nunca se ejecutará. La estructura de repetición `do..while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, *el cuerpo del ciclo siempre se ejecutará cuando menos una vez*. Cuando termina una estructura de repetición `do...while`, la ejecución continúa con la instrucción que va después de la cláusula `while`. No es necesario utilizar llaves en la estructura de repetición `do...while` si sólo hay una instrucción en el cuerpo. Sin embargo, la mayoría de los programadores incluyen las llaves para evitar la confusión entre las instrucciones `while` y `do...while`. Por ejemplo:

```
while ( condición )
```

generalmente es la primera línea de una instrucción `while`. Una instrucción `do...while` sin llaves, alrededor de un cuerpo con una sola instrucción, aparece así:

```
do
   instrucción
while ( condición );
```

http://librosysolucionarios.net



Buena práctica de programación 5.6

Incluya siempre las llaves en una instrucción `do...while`, aun y cuando éstas no sean necesarias. Esto ayuda a eliminar la ambigüedad entre las instrucciones `while` y `do...while` que contienen sólo una instrucción.

El applet de la figura 5.7 utiliza una instrucción `do...while` para dibujar 10 círculos anidados, con el método `drawOval` de la clase `Graphics`.

En el método `paint` (líneas 10 a 22), la línea 14 declara la variable de control `contador` y la inicializa en 1. Al entrar a la instrucción `do...while`, las líneas 17 y 18 llaman al método `drawOval` del objeto `Graphics`. Los cuatro argumentos que representan la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura del *cuadro delimitador* del óvalo (un rectángulo imaginario en el que el óvalo toca el centro de los cuatro lados del rectángulo), se calculan con base en el valor de `contador`. El programa primero dibuja el círculo más interno (que se encuentra en el centro). La esquina superior izquierda del cuadro delimitador para cada círculo subsecuente se acerca cada vez más a la esquina superior izquierda del applet. Al mismo tiempo, la anchura y altura del cuadro delimitador se incrementan para asegurar que cada nuevo círculo contenga a todos los círculos anteriores. La línea 19 incrementa a `contador`; después, el programa evalúa la condición de continuación al final del ciclo.

```
1  // Fig. 5.7: PruebaDoWhile.java
2  // Uso de la instrucción do...while.
3  import java.awt.Graphics;
4
5  import javax.swing.JApplet;
6
7  public class PruebaDoWhile extends JApplet {
8
9     // dibujar líneas en el applet
10    public void paint( Graphics g )
11    {
12       super.paint( g );  // llamar al método paint heredado de JApplet
13
14       int contador = 1;  // inicializar contador
15
16       do {
17          g.drawOval( 110 - contador * 10, 110 - contador * 10,
18             contador * 20, contador * 20 );
19          ++contador;
20       } while ( contador <= 10 );  // fin de instrucción do...while
21
22    } // fin del método paint
23
24 } // fin de la clase PruebaDoWhile
```

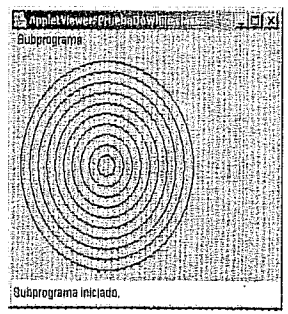


Figura 5.7 Instrucción `do...while`.

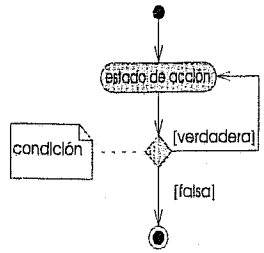


Figura 5.8 Diagrama de actividad de la instrucción de repetición `do...while`.

La figura 5.8 contiene el diagrama de actividad para la instrucción **`do...while`**. Este diagrama aclara que la condición de continuación de ciclo no se evalúa sino hasta que el ciclo ejecute el estado de acción, cuando menos una vez. Compare este diagrama de actividad con el de la instrucción **`while`** (figura 4.5).

5.6 Instrucción de selección múltiple `switch`

En el capítulo 4 hablamos sobre la instrucción **`if`** de selección simple y la instrucción **`if...else`** de selección doble. Java cuenta con la instrucción **`switch`** de selección múltiple para realizar distintas acciones, con base en los posibles valores de una variable o expresión entera. Cada acción se asocia con un valor integral constante (es decir, un valor de tipo **`byte`**, **`short`**, **`int`** o **`char`**, pero no **`long`**) que la variable o expresión pueda asumir. El applet de la figura 5.9 utiliza una instrucción **`switch`** en un método **`paint`** que dibuja líneas, rectángulos u óvalos, con base en un entero que el usuario introduce mediante un cuadro de diálogo de entrada.

```
1  // Fig. 5.9: PruebaSwitch.java
2  // Dibujar líneas, rectángulos u óvalos con base en la entrada del usuario.
3  import java.awt.Graphics;
4
5  import javax.swing.*;
6
7  public class PruebaSwitch extends JApplet {
8     int opcion;  // la opción del usuario en cuanto a cuál figura dibujar
9
10    // inicializar el applet, obteniendo la opción del usuario
11    public void init()
12    {
13       String entrada;  // la entrada del usuario
14
15       // obtener la opción del usuario
16       entrada = JOptionPane.showInputDialog(
17          "Escriba 1 para dibujar líneas\n" +
18          "Escriba 2 para dibujar rectángulos\n" +
19          "Escriba 3 para dibujar óvalos\n" );
20
21       opcion = Integer.parseInt( entrada );  // convertir entrada en int
22
23    } // fin del método init
24
```

Figura 5.9 Instrucción **`switch`** que evalúa múltiples valores de entrada. (Parte 1 de 3.)


```
25    // dibujar figuras en el fondo del applet
26    public void paint( Graphics g )
27    {
28       super.paint( g );  // llamar al método paint heredado de JApplet
29
30       for ( int i = 0; i < 10; i++ ) {  // iterar 10 veces (0-9)
31
32          switch ( opcion ) { // determinar la figura a dibujar
33
34             case 1:  // dibujar una línea
35                g.drawLine( 10, 10, 250, 10 + i * 10 );
36                break;  // fin de procesamiento de case
37
38             case 2:  // dibujar un rectángulo
39                g.drawRect( 10 + i * 10, 10 + i * 10,
40                   50 + i * 10, 50 + i * 10 );
41                break;  // fin de procesamiento de case
42
43             case 3:  // dibujar un óvalo
44                g.drawOval( 10 + i * 10, 10 + i * 10,
45                   50 + i * 10, 50 + i * 10 );
46                break;  // fin de procesamiento de case
47
48             default: // dibujar cadena indicando que se escribió un valor
49                incorrecto g.drawString( "Se escribió un valor incorrecto",
50                   10, 20 + i * 15 );
51
52          } // fin de instrucción switch
53
54       } // fin de instrucción for
55
56    } // fin del método paint
57
58 } // fin de la clase PruebaSwitch
```

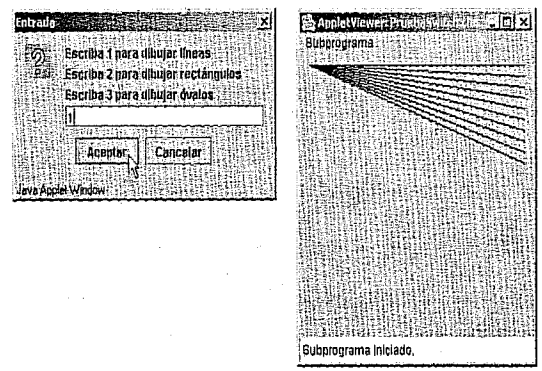


Figura 5.9 Instrucción **`switch`** que evalúa múltiples valores de entrada. (Parte 2 de 3.)

http://librosysdiccionarios.net

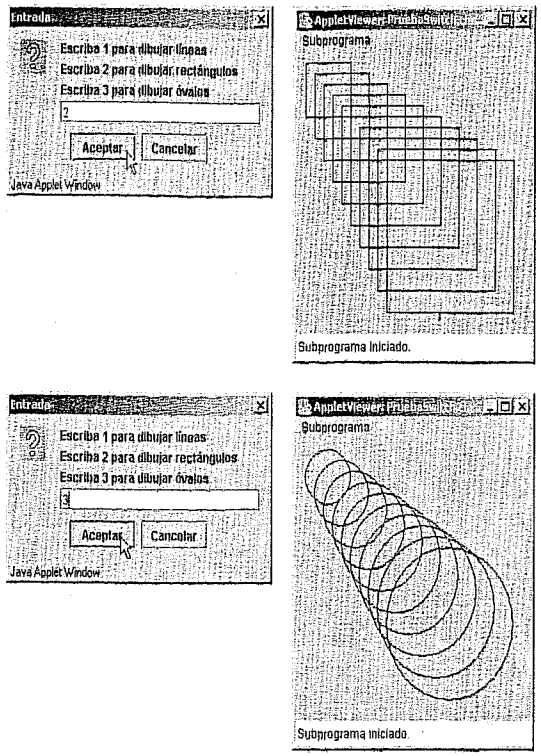


Figura 5.9 Instrucción `switch` que evalúa múltiples valores de entrada. (Parte 3 de 3.)

La línea 8 en el applet `PruebaSwitch` declara el campo `opcion` de tipo `int`. Esta variable guarda la entrada del usuario que determina la figura a dibujar en `paint`.

El método `init` (líneas 11 a 23) declara la variable local `entrada` de tipo `String` en la línea 13. Esta variable almacena la cadena que el usuario escribe en el cuadro de diálogo de entrada. Las líneas 16 a 19 muestran el cuadro de diálogo de entrada con el método estático `showInputDialog` de `JOptionPane`, en donde se pide al usuario que escriba `1` para dibujar líneas, `2` para dibujar rectángulos o `3` para dibujar óvalos. La línea 21 convierte `entrada` en un valor `int` y asigna el resultado al campo `opcion`. Como es un campo de la clase, la variable `opcion` puede usarse en el método `init` y en el método `paint`, en este ejemplo.

El método `paint` (líneas 26 a 56) contiene una instrucción `for` (líneas 30 a 54) que itera 10 veces. En este ejemplo, el encabezado de `for` en la línea 30, utiliza el conteo de base cero. Los valores de `i` para las 10 iteraciones del ciclo son `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` y `9`, y el ciclo termina cuando el valor de `i` se vuelve `10`. (*Nota:* Como lo dijimos en la sección 3.3, el contenedor de applets llama al método `paint` después de los métodos `init` y `start`. El contenedor de applets llama también al método `paint` siempre que el área de la pantalla del applet necesite actualizarse [por ejemplo, después de que otra ventana que cubría el área del applet se haya movido a otra ubicación en la pantalla.])

En el cuerpo de la instrucción `for` se encuentra una instrucción `switch` anidada (líneas 32 a 52), la cual dibuja las figuras con base en un valor entero introducido por el usuario en el método `init`. La instrucción `switch` consiste en un bloque que contiene una secuencia de *etiquetas* `case` y un *caso* `default` opcional.

Cuando el flujo de control llega a `switch`, el programa evalúa la expresión entre paréntesis (`opcion`) que va después de la palabra clave `switch`. A ésta se le llama la *expresión controladora* de `switch`. El programa compara el valor de la expresión controladora (que debe evaluarse como un valor integral de tipo `byte`, `char`, `short`, o `int`) con cada una de las etiquetas `case`. Por ejemplo, si el usuario introduce el entero `2`, el programa compara `2` con cada `case` en el `switch`. Si ocurre una concordancia (`case 2:`), el programa ejecuta las instrucciones de ese `case`. Para el entero `2`, las líneas 38 a 41 dibujan un rectángulo, utilizando cuatro argumentos que representan la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura del rectángulo, y la instrucción `switch` sale inmediatamente con la instrucción `break` (línea 41). Después, el programa incrementa la variable contador en el `for` y vuelve a evaluar la condición de continuación de ciclo, para determinar si debe realizar otra iteración del ciclo.

La instrucción `break` hace que el control del programa pase a la primera instrucción que está después de `switch`. (En este caso, llegamos al final del cuerpo del `for`, por lo que el control fluye hacia la expresión de incremento en el encabezado del `for`.) Sin `break`, las etiquetas `case` en una instrucción `switch` "se pasarían" a las instrucciones en las etiquetas `case` subsecuentes. Por lo tanto, cada vez que ocurriera una concordancia en `switch`, se ejecutarían las instrucciones de todas las etiquetas `case` restantes. (Esta característica es perfecta para programar una canción iterativa.) Si no ocurre una concordancia entre el valor de la expresión controladora y una etiqueta `case`, se ejecuta el caso predeterminado `default` (líneas 48 a 50) y el programa dibuja un mensaje de error en el applet.

Error común de programación 5.8

Olvidar una instrucción `break` cuando es necesaria en un `switch` es un error lógico.

Cada `case` puede tener varias instrucciones. La instrucción `switch` difiere de otras instrucciones de control en cuanto a que no requiere llaves alrededor de varias instrucciones en cada `case`. La figura 5.10 muestra el diagrama de actividad para la instrucción `switch` general. La mayoría de las instrucciones `switch` utilizan una instrucción `break` en cada `case` para terminar la instrucción `switch` después de procesar el `case` correspondiente. La figura 5.10 enfatiza esto, incluyendo instrucciones `break` en el diagrama de actividad. Sin la instrucción `break`, el control no pasaría al final de la instrucción `switch` después de procesar un `case`. En vez de ello, el control pasaría a las acciones del siguiente `case`. El diagrama aclara que la instrucción `break` al final de un `case` ocasiona que el control salga de la instrucción `switch` inmediatamente.

La instrucción `break` no se requiere para el último `case` del `switch` (o para el caso `default`, cuando éste aparece al último), ya que la ejecución continúa con la siguiente instrucción después del `switch`.

Buena práctica de programación 5.7

Debe proporcionar un caso `default` en las instrucciones `switch`. En una instrucción `switch` sin un caso `default`, se ignoran los casos que no tienen concordancia explícita. Al incluir un caso `default`, usted se enfoca en procesar las condiciones excepcionales. Existen situaciones en las que no se necesita el procesamiento predeterminado con `default`.

Buena práctica de programación 5.8

Aunque cada `case` y el caso `default` en una instrucción `switch` pueden ocurrir en cualquier orden, es conveniente colocar el caso `default` al final. Cuando este caso se enlista al último, no se requiere la instrucción `break` para ese `case`. Algunos programadores incluyen esta instrucción `break` por cuestiones de claridad y simetría con los demás casos.

Observe que enlistar etiquetas `case` juntas (como `case 1: case 2: case 3:` sin instrucciones entre los casos) indica que debe realizarse el mismo conjunto de acciones para cada caso.

Al utilizar la instrucción `switch`, recuerde que la expresión después de cada `case` puede ser sólo una *expresión integral constante* (es decir, cualquier combinación de constantes caracteres y enteras, que se evalúe como un valor entero constante). Una constante carácter se representa como el carácter específico entre comillas sencillas, como `'A'` o `'$'`. Una constante entera es simplemente un valor entero. La expresión en cada `case` también puede ser una *variable constante* (es decir, una variable que contenga un valor que no cambie durante todo el programa). Dicha variable se declara con la palabra clave `final` (la cual se describe en el capítulo 6). Cuando hablemos sobre la programación orientada a objetos en el capítulo 9, Programación orientada a objetos: herencia, presentaremos una manera más elegante de implementar la lógica de `switch`. Usaremos una técnica llamada *polimorfismo* para crear programas que sean comúnmente más claros, más fáciles de mantener y de extender que los programas que utilizan la lógica de `switch`.

http://librosysolucionarios.net

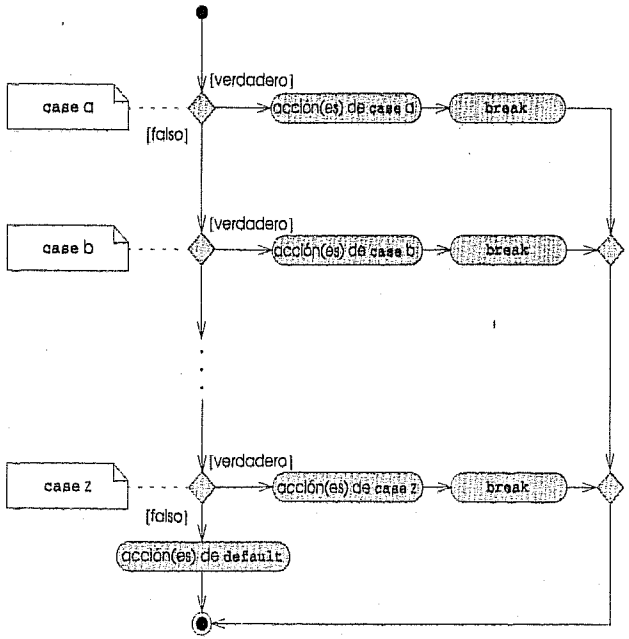


Figura 5.10 Diagrama de actividad de la instrucción `switch` de selección múltiple con instrucciones `break`.

5.7 Instrucciones `break` y `continue`

Además de las instrucciones de selección y repetición, Java proporciona las instrucciones ***`break`*** y ***`continue`*** (que se presentan en ésta y en la sección 5.8) para alterar el flujo de control.

Instrucción `break`

Esta instrucción, al ejecutarse en un ciclo `while`, `for`, `do...while` o `switch`, ocasiona la salida inmediata de esa instrucción. La ejecución continúa con la primera instrucción que va después de la instrucción de control. Los usos comunes de la instrucción `break` son para escapar antes de tiempo de un ciclo, o para omitir el resto de un `switch` (como en la figura 5.9). La figura 5.11 muestra la instrucción `break` en un `for`.

Cuando la instrucción `if` de la línea 14 en la instrucción `for` (líneas 12 a 19) detecta que `cuenta` vale `5`, se ejecuta la instrucción `break` de la línea 15. Esto termina la instrucción `for`, y el programa procede a la línea 21 (inmediatamente después de la instrucción `for`), la cual completa la cadena a mostrar en un cuadro de diálogo de mensaje, en la línea 22. El ciclo ejecuta completamente su cuerpo sólo cuatro veces.

Instrucción `continue`

Esta instrucción, al ejecutarse en un ciclo `while`, `for` o `do...while`, evita las instrucciones restantes en el cuerpo del ciclo y procede con la siguiente iteración del mismo. En las instrucciones `while` y `do... while`, el programa evalúa la condición de continuación de ciclo inmediatamente después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y luego el programa evalúa la condición de continuación de ciclo. En la sección 5.3 dijimos que `while` podía utilizarse, en la mayoría de los casos, en lugar de `for`. La única excepción ocurre cuando la expresión de incremento en el `while` va después de una instrucción `continue`. En este caso, el incremento no se ejecuta antes de que el programa evalúe la condición de continuación de ciclo, por lo que el `while` no se ejecuta de la misma forma que el `for`. La figura 5.12 utiliza la instrucción `continue` en un `for` para evitar la instrucción de concatenación de cadenas


```
1  // Fig. 5.11: PruebaBreak.java
2  // Terminación de un ciclo con break.
3  import javax.swing.JOptionPane;
4
5  public class PruebaBreak {
6
7     public static void main( String args[] )
8     {
9        String salida = "";
10       int cuenta;
11
12       for ( cuenta = 1; cuenta <= 10; cuenta++ ) {  // iterar 10 veces
13
14          if ( cuenta == 5 )  // si cuenta vale 5,
15             break;           // terminar el ciclo
16
17          salida += cuenta + " ";
18
19       } // fin de instrucción for
20
21       salida += "\nSe salió del ciclo en cuenta = " + cuenta;
22       JOptionPane.showMessageDialog( null, salida );
23
24       System.exit( 0 );  // terminar la aplicación
25
26    } // fin de main
27
28 } // fin de la clase PruebaBreak
```

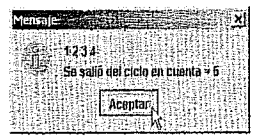


Figura 5.11 Instrucción `break` que provoca la salida de una instrucción `for`.

(línea 16) cuando el `if` (línea 13) determina que el valor de `cuenta` es `5`. Cuando la instrucción `continue` se ejecuta, el control del programa continúa con el incremento de la variable de control en la instrucción `for`.

```
1  // Fig. 5.12: PruebaContinue.java
2  // Continuar con la siguiente iteración de un ciclo.
3  import javax.swing.JOptionPane;
4
5  public class PruebaContinue {
6
7     public static void main( String args[] )
8     {
9        String salida = "";
10
11       for ( int cuenta = 1; cuenta <= 10; cuenta++ ) {  // iterar 10 veces
12
```

Figura 5.12 Instrucción `continue` para terminar una de las iteraciones de una instrucción `for`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
13          if ( cuenta == 5 )  // si cuenta vale 5,
14             continue;        // evitar el código restante en el ciclo
15
16          salida += cuenta + " ";
17
18       } // fin de instrucción for
19
20       salida += "\nSe utilizó continue para evitar imprimir un 5";
21       JOptionPane.showMessageDialog( null, salida );
22
23       System.exit( 0 );  // terminar la aplicación
24
25    } // fin de main
26
27 } // fin de la clase PruebaContinue
```

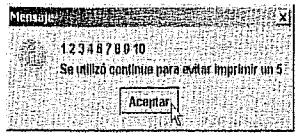


Figura 5.12 Instrucción **continue** para terminar una de las iteraciones de una instrucción **for**. (Parte 2 de 2.)

Tip de rendimiento 5.2

Si se utilizan apropiadamente, las instrucciones **break** *y* **continue** *pueden ser más veloces que las técnicas estructuradas correspondientes. Algunos programadores sienten que* **break** *y* **continue** *violan la programación estructurada. Como los efectos de estas instrucciones pueden lograrse con técnicas de programación estructurada, estos programadores no utilizan* **break** *ni* **continue**.

Observación de ingeniería de software 5.1

Hay una resistencia entre lograr una ingeniería de software con calidad y lograr el software con el mejor rendimiento. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones, excepto las que impliquen un rendimiento intensivo, aplique la siguiente regla empírica: primero haga que su código sea simple y correcto; después haga que sea más rápido y pequeño, pero sólo si es necesario.

5.8 Instrucciones `break` y `continue` etiquetadas

Java proporciona las instrucciones **break** y **continue** etiquetadas para los casos en los que un programa necesita alterar el flujo de control en instrucciones de control anidadas.

Instrucción **break** *etiquetada*

La instrucción **break** que se presentó en la sección 5.7 permite que un programa se salga del ciclo **while**, **for**, **do...while** o **switch** en el cual aparece. En ocasiones estas estructuras de control se anidan en otras instrucciones de repetición. Tal vez un programa necesite salir de toda la instrucción de control anidada en una sola operación, en vez de tener que esperar a que la instrucción de control anidada complete su ejecución de manera normal. Para salir de dichas instrucciones de control anidadas, puede utilizar la *instrucción* **break** *etiquetada*. Esta instrucción, al ejecutarse en un ciclo **while**, **for**, **do...while** o **switch**, provoca la salida inmediata de esa instrucción de control y de cualquier número de instrucciones que contenga. La ejecución del programa continúa con la primera instrucción que va después de la *instrucción etiquetada* adjunta. La instrucción que va después de la etiqueta puede ser una instrucción de repetición o un bloque en el que aparezca una instrucción de repetición. La figura 5.13 muestra el uso de la instrucción **break** etiquetada en una instrucción **for** anidada.

El bloque (líneas 11 a 33 en la figura 5.13) empieza con una *etiqueta* (un identificador seguido de dos puntos) en la línea 11; aquí utilizamos la etiqueta "**alto:**". El bloque va encerrado entre llaves (líneas 11 y 33) e incluye la instrucción **for** anidada (líneas 14 a 28) y la instrucción de concatenación de cadenas en la línea 31.

http://librosysolucionarios.net

Cuando el **if** de la línea 19 detecta que **fila** es igual a **5**, se ejecuta la instrucción **break** de la línea 20. Esta instrucción termina tanto el **for** de las líneas 17 a 24, como el **for** que lo encierra en las líneas 14 a 28. Posteriormente, el programa procede de inmediato a la línea 35, la primera instrucción después del bloque etiquetado. El **for** exterior ejecuta completamente su cuerpo sólo cuatro veces. La instrucción de concatenación de cadenas de la línea 31 nunca se ejecuta, ya que se encuentra en el cuerpo del bloque etiquetado y el **for** exterior nunca se completa.

Buena práctica de programación 5.9

Muchos niveles de instrucciones de control anidadas pueden hacer que un programa sea difícil de leer. Como regla general, evite utilizar más de tres niveles de anidamiento.

```
1  // Fig. 5.13: PruebaBreakEtiqueta.java
2  // Instrucción break etiquetada.
3  import javax.swing.JOptionPane;
4
5  public class PruebaBreakEtiqueta {
6
7     public static void main( String args[] )
8     {
9        String salida = "";
10
11       alto: { // bloque etiquetado
12
13          // contar 10 filas
14          for ( int fila = 1; fila <= 10; fila++ ) {
15
16             // contar 5 columnas
17             for ( int columna = 1; columna <= 5 ; columna++ ) {
18
19                if ( fila == 5 )  // si la fila es 5,
20                   break alto;    // saltar al final del bloque alto
21
22                salida += "*  ";
23
24             } // fin de instrucción for interna
25
26             salida += "\n";
27
28          } // fin de for externo
29
30          // la siguiente línea se evita
31          salida += "\nLos ciclos terminaron normalmente";
32
33       } // fin de bloque etiquetado
34
35       JOptionPane.showMessageDialog( null, salida,
36          "Probando break con una etiqueta",
37          JOptionPane.INFORMATION_MESSAGE );
38
39       System.exit( 0 );  // terminar la aplicación
40
41    } // fin de main
42
43 } // fin de la clase PruebaBreakEtiqueta
```

Figura 5.13 Instrucción **break** etiquetada que provoca la salida de una instrucción **for** anidada. (Parte 1 de 2.)



Figura 5.13 Instrucción **break** etiquetada que provoca la salida de una instrucción **for** anidada. (Parte 2 de 2.)

Instrucción continue etiquetada

La instrucción **continue** procede con la siguiente iteración (repetición) del ciclo **while**, **for** o **do...while** inmediatamente adjunto. La *instrucción* ***continue*** *etiquetada* evita las instrucciones restantes en el cuerpo de esa instrucción, junto con cualquier número de instrucciones de repetición adjuntas, y procede con la siguiente iteración de la *instrucción de repetición etiquetada* adjunta (es decir, un ciclo **for**, **while** o **do...while** precedido por una etiqueta). En las instrucciones **while** y **do...while** etiquetadas, el programa evalúa la condición de continuación de ciclo del ciclo etiquetado inmediatamente después de que se ejecuta la instrucción **continue**. En un **for** etiquetado, la expresión de incremento se ejecuta y se evalúa la condición de continuación de ciclo. La figura 5.14 utiliza una instrucción **continue** etiquetada en un **for** anidado para permitir que la ejecución continúe con la siguiente iteración del **for** exterior.

```
1  // Fig. 5.14: PruebaContinueEtiqueta.java
2  // Instrucción continue etiquetada.
3  import javax.swing.JOptionPane;
4
5  public class PruebaContinueEtiqueta {
6
7     public static void main( String args[] )
8     {
9        String salida = "";
10
11       siguienteFila: // etiqueta de destino de la instrucción continue
12
13          // contar 5 filas
14          for ( int fila = 1; fila <= 5; fila++ ) {
15             salida += "\n";
16
17             // contar 10 columnas por fila
18             for ( int columna = 1; columna <= 10; columna++ ) {
19
20                // si la columna es mayor que la fila, empezar la siguiente
21                fila if ( columna > fila )
22                   continue siguienteFila; // siguiente iteración del ciclo
23                                           etiquetado
24                salida += "*  ";
25
26             } // fin de instrucción for interior
27
28          } // fin de instrucción for exterior
29
```

Figura 5.14 Instrucción **continue** etiquetada para terminar una instrucción **for** anidada. (Parte 1 de 2.)

http://librosysolucionarios.net


```
30        JOptionPane.showMessageDialog( null, salida,
31           "Probando continue con una etiqueta",
32           JOptionPane.INFORMATION_MESSAGE );
33
34        System.exit( 0 );  // terminar la aplicación
35
36     } // fin de main
37
38  } // fin de la clase PruebaContinueEtiqueta
```

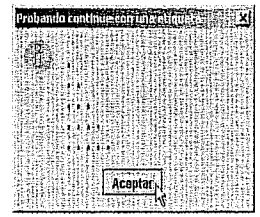


Figura 5.14 Instrucción **continue** etiquetada para terminar una instrucción **for** anidada. (Parte 2 de 2.)

La instrucción **for** etiquetada (líneas 11 a 28) realmente empieza en la etiqueta **siguienteFila**. Cuando el **if** de la línea 21 en el **for** interior (líneas 18 a 26) detecta que **columna** es mayor que **fila**, se ejecuta la instrucción **continue** de la línea 22 y el control del programa continúa con el incremento de la variable de control del ciclo **for** exterior. A pesar de que el **for** interior cuenta de 1 a 10, el número de caracteres * mostrados en una fila nunca se excede del valor de **fila**.

Tip de rendimiento 5.3

El programa de la figura 5.14 puede hacerse más simple y eficiente si se reemplaza la condición en el **for**, *en la línea 18, con* **columna <= fila**, *y si se elimina la instrucción* **if** *en las líneas 21 y 22.*

5.9 Operadores lógicos

Cada una de las instrucciones **if**, **if...else**, **while**, **do...while** y **for** requieren una condición para determinar cómo continuar el flujo de control de un programa. Hasta ahora, hemos estudiado solamente *condiciones simples*, como **cuenta <= 10**, **numero != valorCentinela** y **total > 1000**. Las condiciones simples se expresan en términos de los operadores relacionales **>**, **<**, **>=** y **<=**, de los operadores de igualdad **==** y **!=**, y cada decisión evalúa una condición. Para evaluar múltiples condiciones en el proceso de tomar una decisión, realizamos estas pruebas en instrucciones separadas, o en instrucciones **if** o **if...else** anidadas. Algunas veces, las instrucciones de control requieren condiciones más complejas para determinar el flujo de control de un programa.

Java cuenta con los *operadores lógicos* para que los programadores puedan formar condiciones más complejas, mediante la combinación de condiciones simples. Los operadores lógicos son **&&** (*AND condicional*), **||** (*OR condicional*), **&** (*AND lógico booleano*), **|** (OR incluyente lógico booleano), **^** (*OR excluyente lógico booleano*) y **!** (*NOT lógico*, también llamado *negación lógica*).

Operador AND condicional (&&)

Suponga que deseamos asegurarnos, en cierto punto de un programa, que dos condiciones sean *(ambas)* verdaderas antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador **&&** (AND condicional), como se muestra a continuación:

```
if ( genero == FEMENINO && edad >= 65 )
   ++mujeresTerceraEdad;
```

Esta instrucción `if` contiene dos condiciones simples. Por ejemplo, la condición `genero == FEMENINO` debe evaluarse para determinar si el género de una persona es femenino. La condición `edad >= 65` se evalúa para determinar si una persona es un ciudadano de la tercera edad. La instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

que es verdadera *si y sólo si* ambas condiciones simples son verdaderas. Si la condición combinada es verdadera, el cuerpo de la instrucción `if` incrementa `mujeresTerceraEdad` en `1`. Si alguna de las condiciones es falsa, o si ambas son falsas, el programa no lleva a cabo el incremento. Algunos programadores descubren que la condición combinada, que acabamos de ver, es más legible si se le agregan paréntesis redundantes, como:

```
( genero == FEMENINO ) && ( edad >= 65 )
```

La tabla en la figura 5.15 sintetiza el uso del operador `&&`. Esta tabla muestra las cuatro posibles combinaciones de valores `false` y `true` para *expresión1* y *expresión2*. Dichas tablas se llaman *tablas de verdad*. Java evalúa como `false` o `true` todas las expresiones que incluyen operadores relacionales, de igualdad o lógicos.

Operador condicional OR (||)

Ahora suponga que deseamos asegurarnos que cualquiera de las condiciones, *o* ambas sean verdaderas antes de elegir cierta ruta de ejecución. En este caso utilizamos el operador `||` (OR condicional), como en el siguiente fragmento de un programa:

```
if ( promedioSemestral >= 90 || examenFinal >= 90 )
   System.out.println ( "La calificación del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición `promedioSemestral >= 90` se evalúa para determinar si el estudiante merece una "A" en el curso, debido a un sólido desempeño a lo largo del semestre. La condición `examenFinal >= 90` se evalúa para determinar si el estudiante merece una "A"en el curso debido a un excelente desempeño en el examen final. Entonces la instrucción `if` considera la condición combinada

```
promedioSemestral >= 90  ||  examenFinal >= 90
```

y otorga una "A" al estudiante si se cumple una de las condiciones simples, o ambas. La única ocasión en que *no* se imprime el mensaje `"La calificación del estudiante es A"` es cuando ambas condiciones simples son falsas. La figura 5.16 es una tabla de verdad para el operador OR condicional (`||`). El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha.

expresión1	expresión2	expresión1 && expresión2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 5.15 Tabla de verdad para el operador `&&` (AND condicional).

expresión1	expresión2	expresión1 || expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 5.16 Tabla de verdad para el operador `||` (OR condicional).

http://librosysolucionarios.net

Evaluación de corto circuito en condiciones complejas

Las partes de una expresión que contienen a los operadores `&&` u `||` se evalúan sólo hasta que se sabe si la condición es verdadera o falsa. Por lo tanto, la evaluación de la expresión

```
genero == FEMENINO && edad >= 65
```

se detiene inmediatamente si `genero` no es igual a `FEMENINO` (es decir, toda la expresión es `false`) y continúa si `genero` es igual a `FEMENINO` (es decir, toda la expresión podría aún ser `true` si la condición `edad >= 65` es `true`). Esta característica de las expresiones AND condicional y OR condicional se conoce como *evaluación de corto circuito*.



Error común de programación 5.9

En expresiones que utilizan el operador `&&`, es posible que una condición (a la que llamaremos la condición dependiente) pueda requerir que otra condición sea verdadera para que la evaluación de la condición dependiente tenga valor. En este caso, la condición dependiente debe colocarse después de la otra condición, o podría ocurrir un error.

Operadores AND lógico booleano (&) y OR lógico booleano (|)

Los operadores *AND lógico booleano (`&`) y OR inclusivo lógico booleano (`|`)* funcionan de manera idéntica a los operadores `&&` (AND condicional) y `||` (OR condicional), con una excepción: los operadores lógicos booleanos siempre evalúan ambos de sus operandos (es decir, no realizan una evaluación de corto circuito). Por lo tanto, la expresión

```
genero == 1 & edad >= 65
```

evalúa `edad >= 65` sin importar que `genero` sea igual o no a `1`. Esto es útil si el operando derecho del operador AND lógico booleano u OR inclusivo lógico booleano tiene un *efecto secundario* requerido: la modificación del valor de la variable. Por ejemplo, la expresión

```
cumpleanios == true | ++edad >= 65
```

garantiza que la condición `++edad >= 65` se evalúe. Por lo tanto, la variable `edad` se incrementa en la expresión anterior, sin importar que la expresión total sea `true` o `false`.



Buena práctica de programación 5.10

Por cuestión de claridad, evite las expresiones con efectos secundarios en las condiciones. Estos efectos pueden parecer astutos, pero a menudo ocasionan más problemas que satisfacciones.

OR exclusivo lógico booleano (^)

Una condición que contiene al operador *OR exclusivo lógico booleano (`^`)* es verdadera (`true`) *si y solo si uno de sus operandos es `true` y el otro es `false`*. Si ambos operandos son `true` o ambos son `false`, toda la condición es `false`. La figura 5.17 es una tabla de verdad para el operador OR exclusivo lógico booleano (`^`). También se garantiza que este operador evalúa a sus dos operandos.

Operador de negación lógica (!)

Java cuenta con el operador `!` (negación lógica) para permitir que un programador "invierta" el significado de una condición. A diferencia de los operadores lógicos `&&`, `||`, `&`, `|` y `^`, que son operadores binarios que com-

expresión1	expresión2	expresión1 ^ expresión2
false	false	false
false	true	true
true	false	true
true	true	false

Figura 5.17 Tabla de verdad para el operador ^ (OR exclusivo lógico booleano).

expresión	!expresión
false	true
true	false

Figura 5.18 Tabla de verdad para el operador ! (negación lógica, o NOT lógico).

binan dos condiciones, el operador de negación lógica es un operador unario que tiene una sola condición como operando. El operador de negación lógica se coloca antes de una condición para elegir una ruta de ejecución si la condición original (sin el operador de negación lógica) es `false`, tal como en el segmento de programa

```
if ( ! ( calificacion == valorCentinela ) )
   System.out.println( "La siguiente calificación es " + calificacion );
```

el cual ejecuta la instrucción `println` solamente si `calificación` no es igual a `valorCentinela`. Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador de negación lógica tiene una mayor precedencia que el operador de igualdad.

En la mayoría de los casos, el programador podría evitar el uso de la negación lógica, expresando la condición en forma distinta con un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción anterior puede escribirse también así:

```
if ( calificacion != valorCentinela )
   System.out.println( "La siguiente calificación es " + calificacion );
```

Esta flexibilidad puede ayudar a un programador a expresar una condición de una manera más conveniente. La figura 5.18 es una tabla de verdad para el operador de negación lógica.

Ejemplo sobre el uso de los operadores lógicos

La aplicación de la figura 5.19 muestra el uso de todos los operadores lógicos y operadores lógicos booleanos, produciendo sus tablas de verdad. El programa utiliza la concatenación de cadenas para crear la cadena que se muestra en un objeto `JTextArea`.

En la salida de la figura 5.19, las cadenas "true" y "false" indican `true` y `false` para los operandos en cada una de las condiciones. El resultado de la condición se muestra como `true` o `false`. Observe que al concatenar un valor `boolean` con una cadena, Java automáticamente agrega la cadena "false" o "true", con base en el valor `boolean`.

La línea 10 en `main` crea un objeto `JTextArea`. Los números en los paréntesis indican que el objeto `JTextArea` tiene 17 filas y 20 columnas.[3] La línea 13 declara la referencia `desplazable` del tipo `JScrollPane`, y la inicializa con un nuevo objeto `JScrollPane`. La clase `JScrollPane` (del paquete `javax.swing`) proporciona capacidad de desplazamiento a un componente de la GUI. Un objeto `JScrollPane` se iniciali-

```
1  // Fig. 5.19: OperadoresLogicos.java
2  // Operadores lógicos.
3  import javax.swing.*;
4
5  public class OperadoresLogicos {
6
7     public static void main( String args[] )
8     {
```

Figura 5.19 Operadores lógicos. (Parte 1 de 3.)

3. La anchura en píxeles del objeto `JTextArea` se determina multiplicando el número de columnas especificado por el número de píxeles para la anchura de una letra M mayúscula en el tipo de letra predeterminado del objeto `JTextArea`.


```
9        // crear JTextArea para mostrar los resultados
10       JTextArea areaSalida = new JTextArea( 17, 20 );
11
12       // adjuntar JTextArea a un JScrollPane, para desplazarse por los
13       resultados JScrollPane desplazador = new JScrollPane( areaSalida );
14
15       // crear la tabla de verdad para el operador && (AND condicional)
16       String salida = "AND condicional (&&)" +
17          "\nfalse && false: " + ( false && false ) +
18          "\nfalse && true: " + ( false && true ) +
19          "\ntrue && false: " + ( true && false ) +
20          "\ntrue && true: " + ( true && true );
21
22       // crear la tabla de verdad para el operador || (OR condicional)
23       salida += "\n\nOR condicional (||)" +
24          "\nfalse || false: " + ( false || false ) +
25          "\nfalse || true: " + ( false || true ) +
26          "\ntrue || false: " + ( true || false ) +
27          "\ntrue || true: " + ( true || true );
28
29       // crear la tabla de verdad para el operador & (AND lógico booleano)
30       salida += "\n\nAND lógico booleano (&)" +
31          "\nfalse & false: " + ( false & false ) +
32          "\nfalse & true: " + ( false & true ) +
33          "\ntrue & false: " + ( true & false ) +
34          "\ntrue & true: " + ( true & true );
35
36       // crear la tabla de verdad para el operador | (OR inclusivo lógico booleano)
37       salida += "\n\nOR incluyente lógico booleano (|)" +
38          "\nfalse | false: " + ( false | false ) +
39          "\nfalse | true: " + ( false | true ) +
40          "\ntrue | false: " + ( true | false ) +
41          "\ntrue | true: " + ( true | true );
42
43       // crear la tabla de verdad para el operador ^ (OR excluyente lógico booleano)
44       salida += "\n\nOR excluyente lógico booleano (^)" +
45          "\nfalse ^ false: " + ( false ^ false ) +
46          "\nfalse ^ true: " + ( false ^ true ) +
47          "\ntrue ^ false: " + ( true ^ false ) +
48          "\ntrue ^ true: " + ( true ^ true );
49
50       // crear la tabla de verdad para el operador ! (negación lógica)
51       salida += "\n\nNOT lógico (!)" +
52          "\n!false: " + ( !false ) +
53          "\n!true: " + ( !true );
54
55       areaSalida.setText( salida );  // colocar los resultados en el objeto
            JTextArea
56       JOptionPane.showMessageDialog( null, desplazador,
57          "Tablas de verdad", JOptionPane.INFORMATION_MESSAGE );
58
59       System.exit( 0 );  // terminar la aplicación
60
61    } // fin de main
```

Figura 5.19 Operadores lógicos. (Parte 2 de 3.)

http://librosysolucionarios.net

```
63
64 } // fin de la clase OperadoresLogicos
```

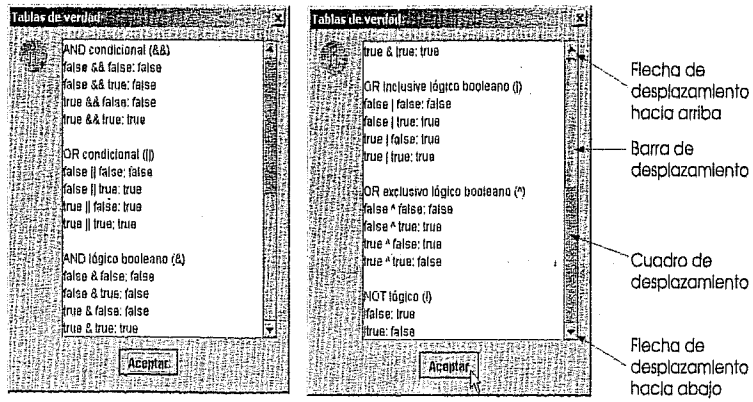


Figura 5.19 Operadores lógicos. (Parte 3 de 3.)

za con el componente GUI al cual le va a proporcionar la capacidad de desplazamiento (en este ejemplo, `areaSalida`). Esta inicialización adjunta el componente de GUI al objeto `JScrollPane`. Cuando usted ejecute esta aplicación, observe la *barra de desplazamiento* en el lado derecho del objeto `JTextArea`. Puede hacer clic en las *flechas* que están en la parte superior o inferior de la barra de desplazamiento, para desplazarse hacia arriba o hacia abajo, respectivamente, a lo largo del texto en el objeto `JTextArea`, una línea a la vez. También puede arrastrar el *cuadro de desplazamiento* (conocido como el *pulgar*) hacia arriba o hacia abajo para desplazarse con rapidez por el texto.

Las líneas 16 a 53 crean la cadena `salida` que se muestra en el objeto `areaSalida`. La línea 55 utiliza el método `setText` para reemplazar el texto en el objeto `areaSalida` con la cadena `salida`. Las líneas 57 y 58 muestran un cuadro de diálogo de mensaje. El segundo argumento, `desplazador`, indica que el `desplazador` (y el objeto `areaSalida` adjunto a éste) debe mostrarse como el mensaje en el cuadro de diálogo de mensaje, el cual ajusta su tamaño automáticamente para alojar el objeto `JTextArea`.

La figura 5.20 muestra la precedencia y la asociatividad de los operadores de Java presentados hasta ahora. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia.

5.10 Resumen sobre programación estructurada

Así como los arquitectos diseñan edificios empleando la sabiduría colectiva de su profesión, los programadores deben diseñar programas empleando la misma sabiduría colectiva. Nuestro campo es más joven que la arquitectura y nuestra sabiduría colectiva es considerablemente menor. Hemos aprendido que la programación estructurada produce programas que, a comparación de los programas no estructurados, son más fáciles de comprender, probar, depurar, modificar e incluso demuestran ser correctos en un sentido matemático.

La figura 5.21 utiliza diagramas de actividad para sintetizar las instrucciones de control de Java. Los estados inicial y final indican el único punto de entrada y de salida de cada instrucción de control. Si se conectan arbitrariamente los símbolos individuales en un diagrama de actividad, se pueden producir programas no estructurados. Por lo tanto, la profesión de la programación ha elegido un conjunto limitado de estructuras de control, que pueden combinarse en sólo dos maneras simples para crear programas estructurados.

Por cuestión de simpleza, sólo se utilizan instrucciones de control de una sola entrada/una sola salida; sólo hay una manera de entrar y una manera de salir de cada instrucción de control. Es sencillo conectar instrucciones de control en secuencia para formar programas estructurados: el estado final de una instrucción de control se conecta al estado inicial de la siguiente instrucción de control. Esto es, las instrucciones de control simples can una después de otra en un programa en secuencia. A esto le llamamos "apilamiento de estructuras de con-

Operadores	Asociatividad	Tipo
`++ --`	derecha a izquierda	postfijo unario
`++ -- + - ! (tipo)`	derecha a izquierda	unario
`* / %`	izquierda a derecha	multiplicativo
`+ -`	izquierda a derecha	aditivo
`< <= > >=`	izquierda a derecha	relacional
`== !=`	izquierda a derecha	igualdad
`&`	izquierda a derecha	AND lógico booleano
`^`	izquierda a derecha	OR excluyente lógico booleano
`|`	izquierda a derecha	OR inclusivo lógico booleano
`&&`	izquierda a derecha	AND condicional
`||`	izquierda a derecha	OR condicional
`?:`	derecha a izquierda	condicional
`= += -= *= /= %=`	derecha a izquierda	asignación

Figura 5.20 Precedencia/asociatividad de los operadores descritos hasta ahora.

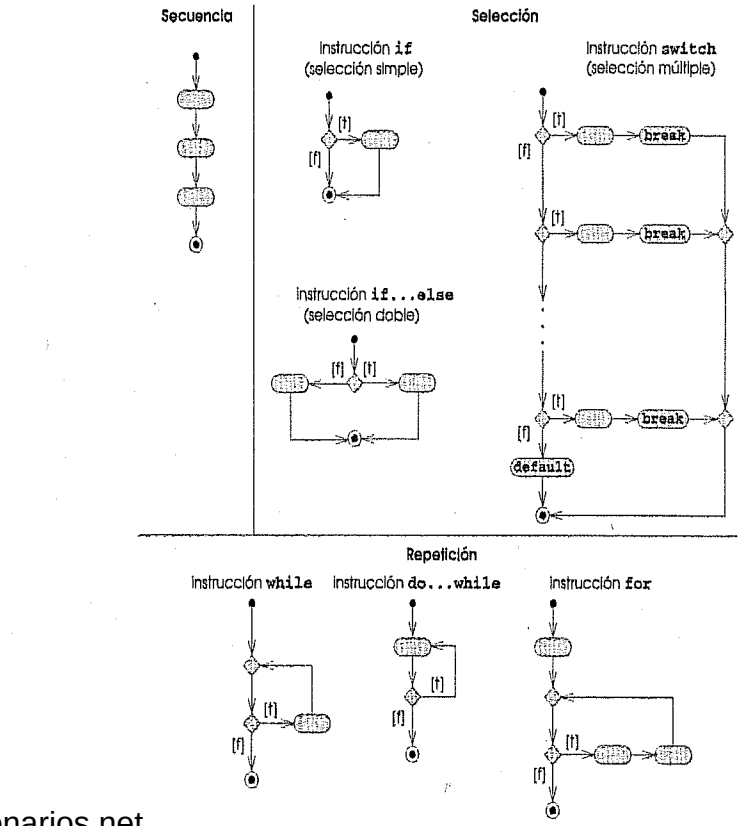


Figura 5.21 Instrucciones de secuencia, selección y repetición de una entrada/una salida de Java.

http://librosysolucionarios.net

trol". Las reglas para formar programas estructurados también permiten el anidamiento de las estructuras de control.

La figura 5.22 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar cualquier acción. Además, las reglas suponen que comenzamos con el diagrama de actividad más sencillo (figura 5.23), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

Al aplicar las reglas de la figura 5.22, siempre se obtiene un diagrama de actividad con una agradable apariencia de bloque de construcción. Por ejemplo, si se aplica la regla 2 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad que contiene muchos estados de acción en secuencia (figura 5.24). La regla 2 genera una pila de estructuras de control, por lo que llamaremos a la regla 2 la *regla de apilamiento*. (*Nota:* Las líneas punteadas verticales en la figura 5.24 no son parte de UML. Las utilizamos para separar los cuatro diagramas de actividad que demuestran cómo se aplica la regla 2 de la figura 5.22.)

Reglas para formar programas estructurados

1) Comenzar con el "diagrama de actividad más sencillo" (figura 5.23).
2) Cualquier estado de acción puede reemplazarse por dos estados de acción en secuencia.
3) Cualquier estado de acción puede reemplazarse por cualquier instrucción de control (secuencia, `if`, `if...else`, `switch`, `while`, `do...while` y `for`).
4) Las reglas 2 y 3 pueden aplicarse tantas veces como se desee y en cualquier orden.

Figura 5.22 Reglas para formar programas estructurados.

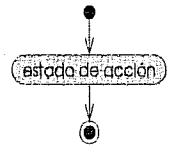


Figura 5.23 El diagrama de actividad más sencillo.

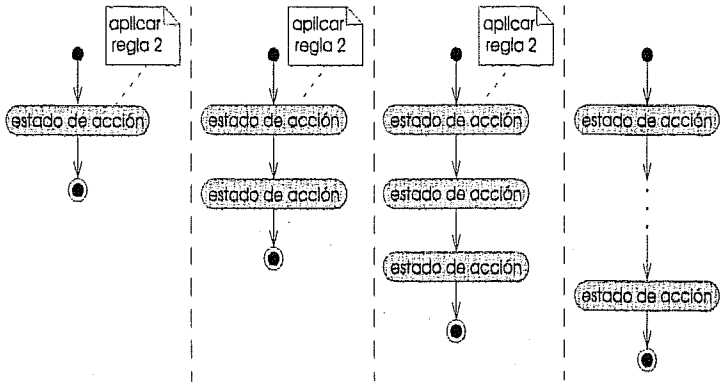


Figura 5.24 El resultado de aplicar la regla 2 de la figura 5.22 repetidamente al diagrama de actividad más sencillo.

http://librosysolucionarios.net

La regla 3 se conoce como la *regla de anidamiento*. Al aplicar la regla 3 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad con instrucciones de control perfectamente apiladas. Por ejemplo, en la figura 5.22 el estado de acción en el diagrama de actividad más sencillo se reemplaza con una instrucción de selección doble (`if...else`). Luego la regla 3 se aplica otra vez a los estados de acción en la instrucción de selección doble, reemplazando cada uno de estos estados de acción con una instrucción de selección doble. Los símbolos punteados de estado de acción, alrededor de cada una de las instrucciones de selección doble, representan el estado de acción que se reemplazó en el diagrama de actividad original, más sencillo. (*Nota:* Las flechas punteadas y los símbolos punteados de estado de acción que se muestran en la figura 5.25, no son parte de UML. Aquí se utilizan como dispositivos pedagógicos para ilustrar que cualquier estado de acción puede reemplazarse con un enunciado de control.)

La regla 4 genera instrucciones más grandes, más implicadas y más profundamente anidadas. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.22, constituyen el conjunto de todos los diagramas de actividad posibles y, por lo tanto, el conjunto de todos los programas estructurados posibles. La belleza de la metodología estructurada es que utilizamos sólo siete instrucciones de control simples de una entrada/una salida, y las ensamblamos en una de sólo dos formas simples.

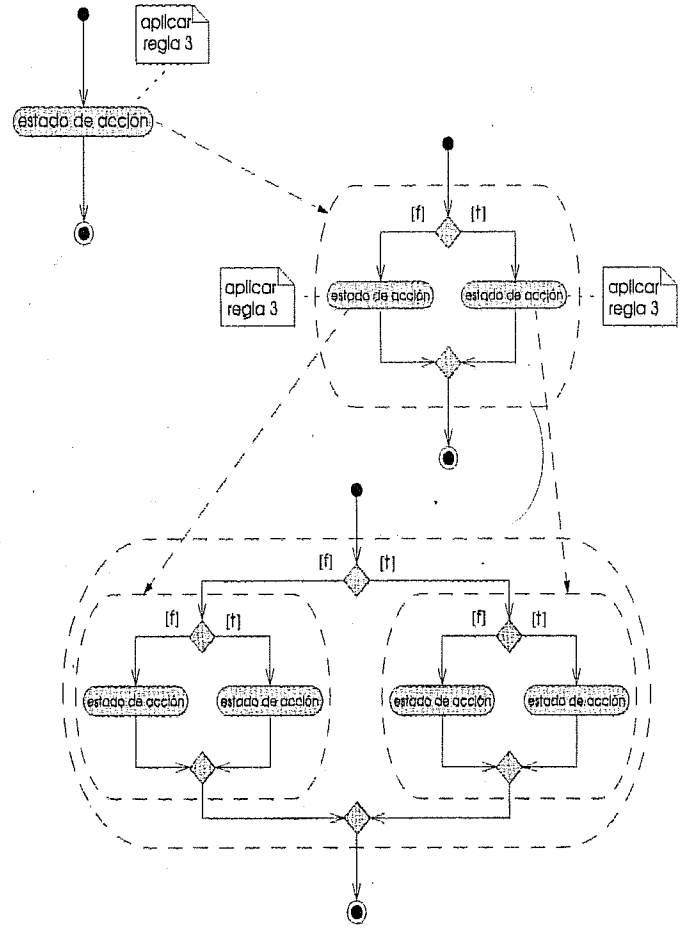


Figura 5.25 Aplicación de la regla 3 de la figura 5.22 al diagrama de actividad más sencillo.

Si se siguen las reglas de la figura 5.22, no podrá crearse un diagrama de actividad con sintaxis ilegal (como el de la figura 5.26). Si usted no está seguro de que cierto diagrama sea legal, aplique las reglas de la figura 5.22 en orden inverso para reducir el diagrama al diagrama de actividad más sencillo. Si el diagrama puede reducirse al diagrama de actividad más sencillo, el diagrama original es estructurado; de lo contrario, no es estructurado.

La programación estructurada promueve la simpleza. Bohm y Jacopini nos han dado el resultado de que sólo se necesitan tres formas de control para implementar un algoritmo:

- Secuencia.
- Selección.
- Repetición.

La estructura de secuencia es trivial. Simplemente enliste las instrucciones a ejecutar en el orden en el que deben ejecutarse.

La selección se implementa en una de tres formas:

- Instrucción `if` (selección simple).
- Instrucción `if...else` (selección doble).
- Instrucción `switch` (selección múltiple).

De hecho, es sencillo demostrar que la instrucción `if` simple es suficiente para ofrecer cualquier forma de selección; todo lo que pueda hacerse con las instrucciones `if...else` y `switch` puede implementarse si se combinan instrucciones `if` (aunque tal vez no con tanta claridad y eficiencia).

La repetición se implementa en una de tres maneras:

- Instrucción `while`.
- Instrucción `do...while`.
- Instrucción `for`.

Es sencillo demostrar que la instrucción `while` es suficiente para proporcionar cualquier forma de repetición. Todo lo que puede hacerse con las instrucciones `do...while` y `for`, puede hacerse también con la instrucción `while` (aunque tal vez no sea tan sencillo).

Si se combinan estos resultados, se demuestra que cualquier forma de control necesaria en un programa en Java puede expresarse en términos de

- Secuencia.
- Instrucción `if` (selección).
- Instrucción `while` (repetición).

y que estos tres elementos pueden combinarse en sólo dos formas: apilamiento y anidamiento. Evidentemente, la programación estructurada fomenta la sencillez.

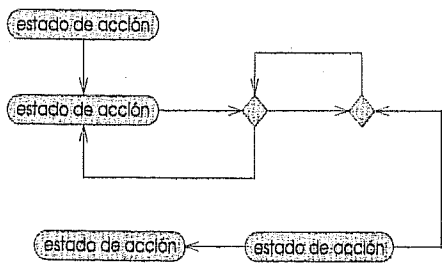


Figura 5.26 Diagrama de actividad con sintaxis ilegal.

En este capítulo vimos cómo crear programas a partir de instrucciones de control que contienen acciones y decisiones. En el capítulo 6 presentaremos otra unidad de estructuración de programas, conocida como *método*. Aprenderemos a crear programas extensos mediante la combinación de métodos que, a su vez, están compuestos de instrucciones de control. También hablaremos sobre cómo los métodos fomentan la reutilización de código. En el capítulo 8 hablaremos más detalladamente sobre la otra unidad de estructuración de programas disponible en Java, conocida como *clase*. Después crearemos objetos a partir de clases y continuaremos con nuestro análisis de la programación orientada a objetos.

5.11 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar los estados y actividades de los objetos

En la sección 4.14 identificamos muchos de los atributos de clase necesarios para implementar el simulador del elevador, y los agregamos al diagrama de clases de la figura 4.18. En esta sección mostraremos cómo esos atributos, aunados a las relaciones de un objeto, representan el *estado* o condición de un objeto. Identificaremos el conjunto de posibles estados que pueden ocupar nuestros objetos, y hablaremos sobre cómo estos objetos cambian de estado en respuesta a los *eventos*. También hablaremos sobre el flujo de trabajo o las *actividades*, que un objeto realiza en nuestra simulación del elevador.

Diagramas de estado

Cada uno de los objetos en un sistema pasa a través de una serie de *estados*. Un estado es la condición de un objeto en un tiempo dado. Los *diagramas de estado* (también conocidos como *diagramas de transición de estado*) proporcionan a los diseñadores una manera de expresar cómo, y bajo qué condiciones, los objetos cambian de estado en un sistema. A diferencia de los diagramas de clases presentados en secciones anteriores del ejemplo práctico, los diagramas de estado modelan una parte del comportamiento del sistema.

La figura 5.27 es un diagrama de estado simple que modela los estados de un objeto, ya sea de la clase `BotonPiso` o `BotonElevador`. UML representa cada estado en un diagrama de estados como un *rectángulo redondeado*, con el nombre del estado colocado dentro del rectángulo. Un *círculo relleno* con una punta de flecha designa el estado inicial. Observe que modelamos esta información de estado como el atributo `oprimido` tipo `Boolean` del diagrama de clases de la figura 4.18. Este atributo se inicializa con `false`, que es el estado "No oprimido", de acuerdo con el diagrama de estados.

Las flechas indican las *transiciones* entre los estados. Un objeto puede pasar de un estado a otro, en respuesta a *eventos* y mensajes. Por ejemplo, los objetos `BotonPiso` y `BotonElevador` cambian del estado "No oprimido" al estado "Oprimido" en respuesta a un evento `botonOprimido`, y el atributo `oprimido` cambia a un valor de `true`. El nombre del evento que produce una transición se escribe cerca de la línea que corresponde a esa transición. (Explicaremos los eventos con más detalle, en las secciones 7.10 y 11.9.)

Modelamos las transiciones de estado de otras clases, como `Luz`, `Elevador` y `Persona`, con diagramas de estados similares. La clase `Luz` tiene un estado "encendido" y un estado "apagado"; las transiciones entre estos estados ocurren como resultado de mensajes "encender" y "apagar", respectivamente. La clase `Elevador` y la clase `Persona` tienen cada una un estado "en movimiento" y un estado "en espera"; las transiciones entre estos estados ocurren como resultado de mensajes "empezar a moverse" y "dejar de moverse", respectivamente.

Diagramas de actividad

Al igual que un diagrama de estados, un *diagrama de actividad* modela los aspectos del comportamiento del sistema. A diferencia del diagrama de estados, el diagrama de actividad modela el *flujo de trabajo* (secuencia

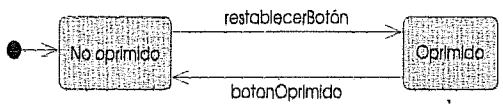


Figura 5.27 Diagrama de estados para los objetos `BotonPiso` y `BotonElevador`.

http://librosysolucionarios.net

de actividades) de un objeto durante la ejecución de los programas. Un diagrama de actividad es un diagrama de flujo que modela las *acciones* que el objeto va a realizar, y en qué orden. El diagrama de actividad de la figura 5.28 modela las actividades de una persona. El diagrama empieza con la persona avanzando hacia el botón del piso. Si la puerta del piso está abierta, la persona espera a que salga el pasajero actual (si existe uno) y luego entra al elevador,[4] Si la puerta del piso está cerrada, la persona oprime el botón del piso y espera a que el elevador abra la puerta. Al abrirse la puerta, la persona espera a que salga el pasajero del elevador (si existe uno) y luego entra al elevador. La persona oprime el botón del elevador, el cual hace que el elevador se mueva hacia el otro piso. Entonces la persona espera a que las puertas se vuelvan a abrir y sale del elevador, una vez que las puertas se abren.

UML representa las actividades como óvalos en los diagramas de actividad. El nombre de la actividad se coloca dentro del óvalo. Una flecha conecta dos actividades, indicando el orden en el que éstas se realizan. El círculo relleno indica la actividad inicial. En este caso, la primera persona pasa al estado "avanzando hacia el botón del piso". El flujo de actividad llega a una *rama* (una bifurcación indicada por el *símbolo de rombo pequeño*) después de que la persona avanza al botón del piso. Este punto determina la siguiente actividad, con base en la *condición de guardia* asociada (encerrada entre corchetes, a un lado de la transición), la cual establece que la transición debe ocurrir si se cumple la condición. Por ejemplo, en la figura 5.28, si la puerta del piso está cerrada, la persona oprime el botón del piso, espera a que se abra la puerta, espera a que salga el pasajero (si hay uno) y luego entra al elevador. No obstante, si la puerta del piso ya está abierta, la persona espera

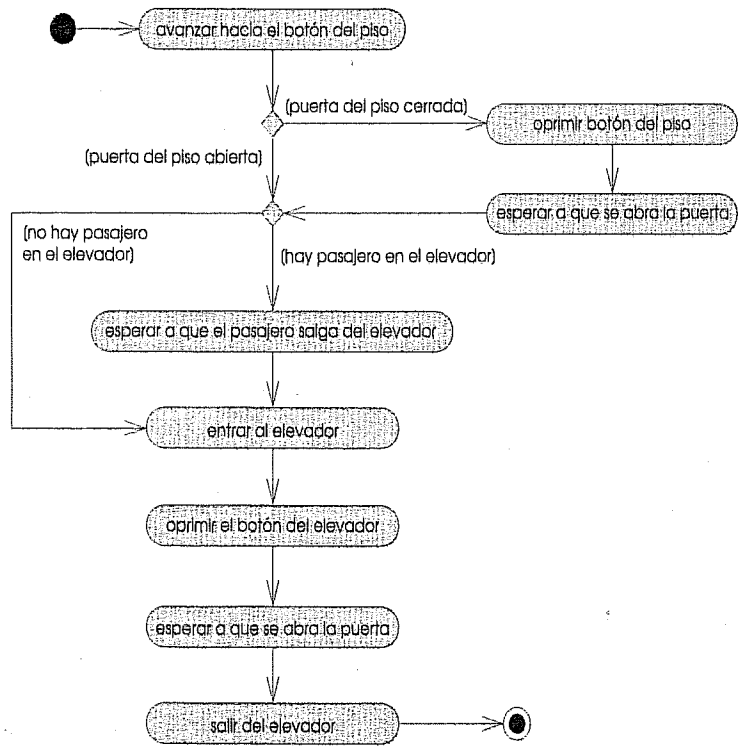


Figura 5.28 Diagrama de actividad para un objeto `Persona`.

4. Utilizaremos el *subprocesamiento múltiple* y los *métodos* `synchronized` en la sección 16.11, para garantizar que el pasajero salga antes de que entre la persona que está esperando.

a que el pasajero (si hay uno) salga del elevador, y luego entra a éste. Sin importar que la puerta del piso haya estado abierta o cerrada en el último símbolo de rombo pequeño, la persona ahora oprime el botón del elevador (el cual hace que las puertas se cierren y que el elevador se mueva hacia el otro piso) y espera a que la puerta del elevador se abra; cuando ésta se abre, la persona sale del elevador. El círculo relleno encerrado dentro de una circunferencia indica el fin de la actividad. Utilizamos diagramas de actividad para demostrar el flujo de control para las instrucciones de control presentadas en los capítulos 4 y 5.

La figura 5.29 muestra un diagrama de actividad para el elevador. Después de la actividad inicial, el diagrama comienza cuando se oprime un botón. Si el botón es un botón del elevador, éste asigna el valor falso a `llamado` (una variable `Boolean` modelada originalmente en la figura 4.18), de manera que el elevador no regresará a ese piso a menos que el botón se oprima otra vez. Después de esto, el elevador cierra su puerta, se mueve al otro piso, restablece el botón del elevador, suena el timbre y abre la puerta del elevador. Si el botón es un botón del piso, la rama determina la siguiente transición, dependiendo de si el elevador se está moviendo o no. Si el elevador está inmóvil, la siguiente rama determina cuál de los botones de piso generó la solicitud. Si la solicitud se originó desde el piso en el que el elevador se encuentra actualmente, el elevador restablece su botón, suena el timbre y abre la puerta. Si la solicitud se originó desde el piso opuesto, el elevador cierra la puerta y se mueve al piso opuesto (destino), en donde restablece su botón, suena el timbre y abre su puerta. Ahora considere la actividad si el elevador se está moviendo. Una rama separada determina cuál botón de piso generó la solicitud. Si la solicitud se originó desde el piso de destino, el elevador continúa viajando a ese piso. Si la solicitud se originó desde el piso del que partió el elevador, éste continúa viajando al piso de destino, pero no debe olvidarse de regresar al piso que lo solicitó. El atributo `llamado` se establece en `true`, de manera que el elevador sepa cómo regresar al otro piso, después de abrir su puerta.

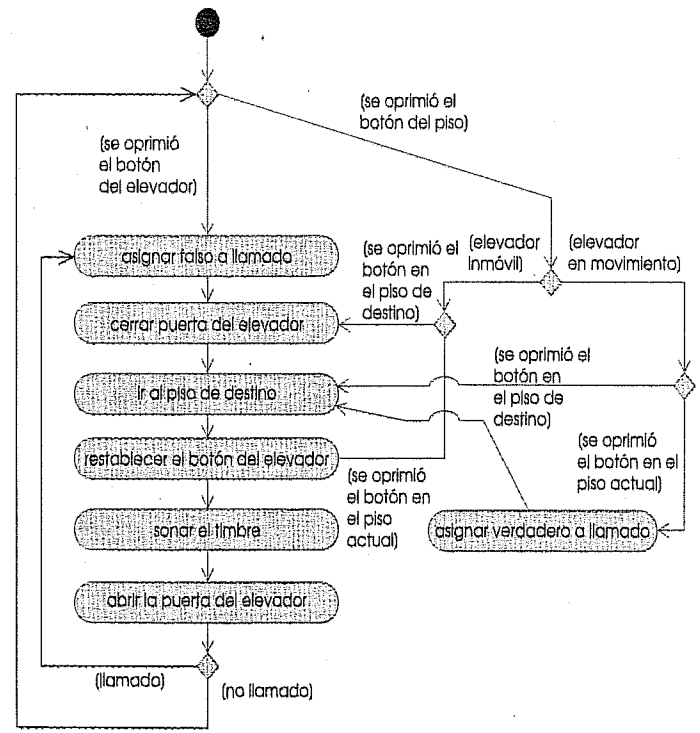


Figura 5.29 Diagrama de actividad para el objeto `Elevador`.

http://librosysolucionarios.net

Hemos realizado los primeros pasos para modelar el comportamiento del sistema, y hemos demostrado cómo participan los atributos de un objeto en la actividad de ese objeto. En la sección 6.17 investigaremos los comportamientos de todas las clases, para dar una interpretación más precisa del comportamiento del sistema al "llenar" el tercer compartimiento para las clases en nuestro diagrama de clases.

RESUMEN

- La instrucción `for` se encarga de todos los detalles acerca de la repetición controlada por contador. El formato general de la instrucción `for` es

  `for (` *inicialización*`;` *condiciónDeContinuaciónDeCiclo*`;` *incremento* `)`
  instrucción

  en donde la expresión de *inicialización* inicializa la variable de control del ciclo, *condicionDeContinuaciónDeCiclo* es la condición que determina si el ciclo debe continuar ejecutándose, e *incremento* modifica la variable de control, de manera que la condición de continuación de ciclo eventualmente se vuelva falsa.
- La clase `NumberFormat` se encuentra en el paquete `java.text`. El método estático `getCurrencyInstance` de `NumberFormat` devuelve un objeto `NumberFormat` que puede dar formato a los números como valores monetarios. El argumento `Locale.US` indica que los valores monetarios deben mostrarse empezando con un signo de dólares ($), se debe usar un punto decimal para separar los dólares y los centavos, y se debe usar una coma para separar los miles. La clase `Locale` se encuentra en el paquete `java.util`.
- Un objeto `JTextArea` es un componente de GUI capaz de mostrar muchas líneas de texto.
- El método `setText` reemplaza el texto en un objeto `JTextArea`. El método `append` agrega texto al final del texto en un objeto `JTextArea`.
- El mensaje que se muestra con el método `showMessageDialog` de `JOptionPane` puede ser una cadena o un componente de GUI, como un objeto `JTextArea`.
- La instrucción `switch` maneja una serie de decisiones, en las que cierta variable o expresión se evalúa con respecto a los valores que puede asumir, y se toman distintas acciones en base a esto. En la mayoría de los programas, es necesario incluir una instrucción `break` después de las instrucciones para cada `case`. Varias instrucciones `case` pueden ejecutar las mismas instrucciones, si las etiquetas `case` se enlistan en conjunto antes de las instrucciones. La instrucción `switch` puede evaluar sólo expresiones integrales constantes de los tipos `byte`, `short`, `int` o `char`, pero no `long`.
- La instrucción `do...while` evalúa la condición de continuación de ciclo al final del ciclo, por lo que el cuerpo de éste se ejecuta cuando menos una vez. El formato de la instrucción `do...while` es

  ```
  do {
     instrucción
  } while ( condición );
  ```

- La instrucción `break`, cuando se ejecuta en una de las instrucciones de repetición, provoca la salida inmediata de la instrucción de repetición.
- La instrucción `continue`, cuando se ejecuta en una de las instrucciones de repetición, evita cualquier instrucción restante en el cuerpo de la instrucción de repetición, y procede con la siguiente iteración del ciclo.
- Para salir de un conjunto anidado de instrucciones de control, utilice la instrucción `break` etiquetada. Esta instrucción, al ejecutarse en una instrucción `while`, `for`, `do...while` o `switch`, provoca la salida inmediata de esa instrucción y de cualquier número de instrucciones adjuntas. La ejecución del programa continúa con la primera instrucción después del bloque etiquetado adjunto.
- La instrucción `continue` etiquetada evita las instrucciones restantes en el cuerpo de una instrucción de repetición y de cualquier número de instrucciones de repetición adjuntas, y procede con la siguiente iteración de la instrucción de repetición etiquetada adjunta.
- Pueden utilizarse operadores lógicos para formar condiciones complejas, combinando las condiciones. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (negación lógica).
- La clase `JScrollPane` proporciona un componente de GUI con funcionalidad de desplazamiento.

TERMINOLOGÍA

`|`, operador
`&&`, operador
`&`, operador
`^`, operador
`||`, operador
`!`, operador
AND condicional (`&&`)
AND lógico booleano (`&`)
`append`, método de `JTextArea`
barra de desplazamiento
`break`
`case`, etiqueta
condición de continuación de ciclo
`continue`
cuadro de desplazamiento
`default`, caso en `switch`
`do...while`, instrucción de repetición
error por desplazamiento en uno
estructuras de control apiladas
evaluación de corto circuito
`for`, instrucción de repetición
`format`, método de la clase `NumberFormat`
`getCurrencyInstance`, método de `NumberFormat`
instrucción `break` etiquetada
instrucción `continue` etiquetada
instrucción de repetición
instrucción de repetición etiquetada
instrucción etiquetada
instrucciones de control anidadas
instrucciones de control de una sola entrada/una sola salida
`java.text`, paquete
`java.util`, paquete
`JScrollPane`, clase
`JTextArea`, clase
`Locale`, clase
Locale.US
negación lógica (`!`)
`NumberFormat`, clase
operadores lógicos
OR condicional (`||`)
OR exclusivo lógico booleano (`^`)
OR inclusivo lógico booleano (`|`)
pulgar de una barra de desplazamiento
repetición controlada por contador
repetición definida
selección múltiple
`switch`, instrucción de selección

EJERCICIOS DE AUTOEVALUACIÓN

5.1 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) El caso `default` es requerido en la instrucción de selección `switch`.
b) La instrucción `break` es requerida en el último caso de una instrucción de selección `switch`.
c) La expresión `( x > y && a < b )` es verdadera si `x > y` es verdadero y si cualquiera de sus operandos, o ambos, son verdaderos.

5.2 Escriba una instrucción o un conjunto de instrucciones en Java, para realizar cada una de las siguientes tareas:
a) Sumar los enteros impares entre 1 y 99, utilizando una instrucción `for`. Suponga que se han declarado las variables enteras `suma` y `cuenta`.
b) Calcular el valor de `2.5` elevado a la potencia de `3`, utilizando el método `pow`.
c) Imprimir los enteros del 1 al 20, utilizando un ciclo `while` y la variable contador `i`. Suponga que la variable `i` se ha declarado, pero no se ha inicializado. Imprimir solamente cinco enteros por línea. [*Sugerencia*: Use el cálculo `i % 5`. Cuando el valor de esta expresión sea 0, imprima un carácter de nueva línea; de lo contrario, imprima un carácter de tabulación. Suponga que este código es una aplicación; utilice el método `System.out.println()` para producir el carácter de nueva línea, y el método `System.out.println( '\t' )` para producir el carácter de tabulación.]

5.3 Encuentre el error en cada uno de los siguientes segmentos de código, y explique cómo corregirlo:
a)
```
i = 1;
while ( i <= 10 );
i++;
}
```
b)
```
for ( k = 0.1; k != 1.0; k += 0.1 )
System.out.println ( k );
```
c)
```
switch ( n ) {
   case 1:
      System.out.println( "El número es 1" );
   case 2:
      System.out.println( "El número es 2" );
      break;
   default:
System.out.println( "El número no es 1 ni 2" );
break;
}
```
d) El siguiente código debe imprimir los valores 1 a 10:
```
n = 1;
```

http://librosysolucionarios.net

```
while ( n < 10 )
System.out.println( n++ );
```

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

5.1 a) Falso. El caso `default` es opcional. Si no se necesita una acción predeterminada, entonces no hay necesidad de un caso `default`. b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no se requiere para el último caso en una instrucción `switch`. c) Falso. Ambas expresiones relacionales deben ser verdaderas para que toda la expresión sea verdadera, cuando se utilice el operador `&&`. d) Verdadero.

5.2 a)
```
suma = 0;
for ( cuenta = 1; cuenta <= 99; cuenta += 2 )
   suma += cuenta;
```
b)
```
Math.pow( 2.5, 3 )
```
c)
```
i = 1;

while ( i <= 20 ) {
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );

   ++i;
}
```
d)
```
for ( i = 1; i <= 20; i++ ) {
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );
}
```
o
```
for ( i = 1; i <= 20; i++ )

   if ( i % 5 == 0 )
      System.out.println( i );
   else
      System.out.println( i + "\t" );
```

5.3 a) Error: El punto y coma después del encabezado `while` provoca un ciclo infinito, y falta una llave izquierda.
Corrección: Reemplazar el punto y coma por una llave izquierda ({), o eliminar tanto el punto y coma (;) como la llave derecha (}).

b) Error: Utilizar un número de punto flotante para controlar una instrucción `for` tal vez no funcione, ya que los números de punto flotante se representan sólo aproximadamente en la mayoría de las computadoras.
Corrección: Utilice un entero, y realice el cálculo apropiado en orden para obtener los valores deseados:

```
for ( k = 1; k != 10, k++ )
System.out.println( ( float ) k / 10 );
```

c) Error: El código que falta es la instrucción `break` en las instrucciones del primer `case`.
Corrección: Agregue una instrucción `break` al final de las instrucciones para el primer `case`. Observe que esta omisión no es necesariamente un error, si el programador desea que la instrucción del `case 2:` se ejecute siempre que lo haga la instrucción del `case 1:`.

d) Error: Se está utilizando un operador relacional inadecuado en la condición de continuación de la instrucción de repetición `while`.
Corrección: Use `<=` en vez de `<`, o cambie `10` a `11`.

EJERCICIOS

5.4 Encuentre el(los) error(es) en cada uno de los siguientes fragmentos de código:

a)
```
For ( i = 100, i >= 1, i++ )
   System.out.print ( i );
```
b) El siguiente código debe imprimirse sin importar que el `valor` entero sea par o impar:
```
switch ( valor % 2 ) {

   case 0:
      System.out.println( "Entero par" );

   case 1:
      System.out.println( "Entero impar" );
}
```
c) El siguiente código debe imprimir los enteros impares del 19 al 1:
```
for ( i = 19; i >= 1; i += 2 )
   System.out.println( i );
```
d) El siguiente código debe imprimir los enteros pares del 2 al 100:
```
contador = 2;

do {
   System.out.println( contador );
   contador += 2;
} While ( contador < 100 );
```

5.5 ¿Qué es lo que hace el siguiente programa?

```
1  public class Imprimir {
2
3     public static void main( String args[] )
4     {
5        for ( int i = 1; i <= 10; i++ ) {
6
7           for (int j = 1; j <= 5; j++ )
8              System.out.print( '@' );
9
10          System.out.println();
11
12       } // fin del for exterior
13
14    }  // fin de main
15
16 } // fin de la clase Imprimir
```

5.6 Escriba una aplicación que encuentre el menor de varios enteros. Suponga que el primer valor leído especifica el número de valores que el usuario va a introducir.

5.7 Escriba una aplicación que calcule el producto de los enteros impares del 1 al 15, y que muestre los resultados en un cuadro de diálogo de mensaje.

5.8 Los *factoriales* se utilizan frecuentemente en los problemas de probabilidad. El factorial de un entero positivo *n* (se escribe como *n!*) es igual al producto de los enteros positivos del 1 a *n*. Escriba una aplicación que evalúe los factoriales de los enteros del 1 al 5. Muestre los resultados en formato tabular, en un objeto `JTextArea` que aparezca en un cuadro de diálogo de mensaje. ¿Qué dificultad podría impedir que usted calculara el factorial de 20?

5.9 Modifique la aplicación de interés compuesto de la figura 5.6, repitiendo sus pasos para las tasas de interés del 5, 6, 7, 8, 9 y 10%. Use un ciclo `for` para variar la tasa de interés. Agregue la capacidad de desplazamiento al objeto `JTextArea`, de manera que pueda desplazarse por todas las líneas de salida.

http://librosysolucionarios.net

5.10 Escriba una aplicación que muestre los siguientes patrones por separado, uno debajo del otro. Use ciclos `for` para generar los patrones. Todos los asteriscos (`*`) deben imprimirse mediante una sola instrucción de la forma `System.out.print( '*' );` la cual hace que los asteriscos se impriman uno al lado del otro. Puede utilizarse una instrucción de la forma `System.out.println();` para posicionarse en la siguiente línea. Puede usarse una instrucción de la forma `System.out.print( ' ' );` para mostrar un espacio para los últimos dos patrones. No debe haber ninguna otra instrucción de salida en el programa. [*Sugerencia:* Los últimos dos patrones requieren que cada línea empiece con un número apropiado de espacios en blanco.]

(a)
```
*
**
***
****
*****
******
*******
********
*********
**********
```

(b)
```
**********
*********
********
*******
******
*****
****
***
**
*
```

(c)
```
**********
*********
********
*******
******
*****
****
***
**
*
```

(d)
```
          *
         **
        ***
       ****
      *****
     ******
    *******
   ********
  *********
 **********
```

5.11 Una aplicación interesante de las computadoras es dibujar gráficos convencionales y de barra (algunas veces denominados "histogramas"). Escriba un applet que lea cinco números, cada uno entre 1 y 30. Por cada número leído, su programa debe dibujar ese número de asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe mostrar *******. [*Sugerencia:* Todos los asteriscos en una línea deben dibujarse con la misma coordenada *y*.]

5.12 Un almacén de pedidos por correo vende cinco productos cuyos precios de venta son los siguientes: Producto 1, $2.98; producto 2, $4.50; producto 3, $9.98; producto 4, $4.49 y producto 5, $6.87. Escriba una aplicación que lea una serie de pares de números, como se muestra a continuación:

a) número del producto;
b) cantidad vendida.

Su programa debe utilizar una instrucción `switch` para determinar el precio de venta de cada producto. Debe calcular y mostrar el valor total de venta de todos los productos vendidos. Use un ciclo controlado por centinela para determinar cuándo debe el programa dejar de iterar para mostrar los resultados.

5.13 Modifique la aplicación de la figura 5.6, de manera que se utilicen sólo enteros para calcular el interés compuesto. [*Sugerencia:* Trate todas las cantidades monetarias como números integrales de centavos. Luego "divida" el resultado en su porción de dólares y su porción de centavos, utilizando las operaciones de división y residuo, respectivamente. Inserte un punto entre las porciones de dólares y centavos.]

5.14 Suponga que `i = 1, j = 2, k = 3` y `m = 2`. ¿Qué es lo que imprime cada una de las siguientes instrucciones?

a) `System.out.println( i == 1 );`
b) `System.out.println( j == 3 );`
c) `System.out.println( i >= 1 && j < 4 );`
d) `System.out.println( m <= 99 & k < m );`
e) `System.out.println( j >= i || k == m );`
f) `System.out.println( k + m < j | 3 - j >= k );`
g) `System.out.println( !( k > m ) );`

5.15 Escriba una aplicación que imprima una tabla de los equivalentes: binario, octal y hexadecimal de los números decimales, en el rango de 1 a 256. Si usted no está familiarizado con estos sistemas numéricos, primero lea el apéndice C. Coloque los resultados en un objeto `JTextArea` con funcionalidad de desplazamiento. Muestre el objeto `JTextArea` en un cuadro de diálogo de mensaje.

5.16 Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Imprima una tabla que muestre el valor aproximado de π, calculando un término de esta serie, dos términos, tres, etcétera. ¿Cuántos términos de esta serie tiene que utilizar para obtener 3.14? ¿3.141? ¿3.1415? ¿3.14159?

5.17 *(Triples de Pitágoras.)* Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para las longitudes de los lados de un triángulo recto se conoce como triple de Pitágoras. Las longitudes de los tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Escriba una aplicación para encontrar todos los triples de Pitágoras para `lado1`, `lado2`, y la `hipotenusa`, que no sean mayores de 500. Use un ciclo `for` triplemente anidado para probar todas las posibilidades. Este método es un ejemplo de la computación de "fuerza bruta". En cursos de ciencias computacionales más avanzados aprenderá que existen muchos problemas interesantes para los cuales no hay otra metodología algorítmica conocida, más que el uso de la fuerza bruta.

5.18 Modifique el ejercicio 5.10 para combinar su código de los cuatro triángulos separados de asteriscos, de manera que los cuatro patrones se impriman uno al lado del otro. Utilice astutamente los ciclos `for` anidados.

```
*          **********  **********          *
**         *********   *********          **
***        ********    ********          ***
****       *******     *******          ****
*****      ******      ******          *****
******     *****       *****          ******
*******    ****        ****          *******
********   ***         ***          ********
*********  **          **          *********
********** *           *          **********
```

5.19 (*Leyes de De Morgan.*) En este capítulo hemos hablado sobre los operadores lógicos `&&`, `&`, `||`, `|`, `^` y `!`. Las leyes de De Morgan pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes establecen que la expresión `!` (*condición1* `&&` *condición2*) es lógicamente equivalente a la expresión (`!`*condición1* `||` `!`*condición2*). También establecen que la expresión `!`(*condición1* `||` *condición2*) es lógicamente equivalente a la expresión (`!`*condición1* `&&` `!`*condición2*). Use las leyes de De Morgan para escribir expresiones equivalentes para cada una de las siguientes expresiones, luego escriba una aplicación que demuestre que, tanto la expresión original como la nueva expresión, producen el mismo valor:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.20 Escriba una aplicación que imprima la siguiente figura de rombo. Puede utilizar instrucciones de salida que impriman un solo asterisco (`*`), un solo espacio o un solo carácter de nueva línea. Maximice el uso de la repetición (con instrucciones `for` anidadas), y minimice el número de instrucciones de salida.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.21 Modifique la aplicación que escribió en el ejercicio 5.20, para que lea un número impar en el rango de 1 a 19, de manera que especifique el número de filas en el rombo. Su programa debe entonces mostrar un rombo del tamaño apropiado.

5.22 Una crítica de las instrucciones `break` y `continue` es que ninguna es estructurada. En realidad, las instrucciones `break` y `continue` pueden reemplazarse en todo momento por instrucciones estructuradas, aunque hacerlo podría ser inadecuado. Describa, en general, cómo eliminaría las instrucciones `break` de un ciclo en un programa, para reemplazarlas con alguna de las instrucciones estructuradas equivalentes. [*Sugerencia:* La instrucción `break` se sale de un ciclo desde el cuerpo de éste. La otra forma de salir es que falle la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo una segunda prueba que indique una "salida anticipada debido a una condición de 'interrupción'".] Use la técnica que desarrolló aquí para eliminar la instrucción `break` de la aplicación de la figura 5.11.

http://librosysolucionarios.net

5.23 ¿Qué hace el siguiente fragmento de programa?

```
for ( i = 1; i <= 5; i++ ) {

   for ( j = 1; j <= 3; j++ ) {

      for ( k = 1; k <= 4; k++ )
         System.out.print( '*' );

      System.out.println();
   } // fin del for interior

   System.out.println();

} // fin del for exterior
```

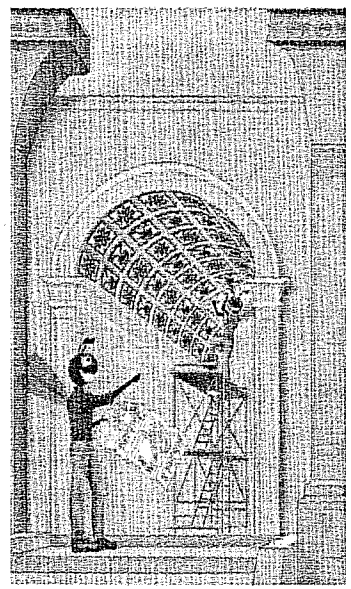
5.24 Describa, en general, cómo eliminaría las instrucciones **continue** de un ciclo en un programa, para reemplazarlas con uno de sus equivalentes estructurados. Use la técnica que desarrolló aquí para eliminar la instrucción **continue** del programa de la figura 5.12.

5.25 Escriba una aplicación que utilice instrucciones de repetición y **switch** para imprimir una canción que contenga elementos iterativos. En el sitio Web **www.12days.com/library/carols/12daysofxmas.htm** encontrará la letra completa de la canción "The twelve days of Christmas" (Los doce días de Navidad) ideal para este ejercicio. Una instrucción **switch** debe utilizarse para imprimir el día (es decir, "Primer", "Segundo", etc.). Una instrucción **switch** separada debe utilizarse para imprimir el resto de cada verso.

6

Métodos

Objetivos



- Aprender a crear programas en forma modular, a partir de pequeñas piezas de código llamadas *métodos*.
- Introducir los métodos matemáticos comunes disponibles en la API de Java.
- Aprender a crear nuevos métodos.
- Conocer los mecanismos para pasar información entre métodos.
- Introducir las técnicas de simulación que se utilizan para generar números aleatorios.
- Comprender cómo la visibilidad de las declaraciones está limitada a regiones específicas de programas.
- Aprender a escribir y utilizar métodos que se llamen a sí mismos.

La forma siempre sigue a la función.
Louis Henri Sullivan

E pluribus unum.
(Uno compuesto de muchos)
Virgil (Publius Vergilius Maro)

¡Oh! volvió a llamar ayer, ofreciéndome volver.
William Shakespeare

Llámenme Ismael
Herman Melville

Cuando me llames así, sonríe.
Owen Wister

http://librosysolucionarios.net

Plan general

6.1 Introducción
6.2 Módulos de programas en Java
6.3 Métodos de la clase `Math`
6.4 Declaraciones de métodos
6.5 Promoción de argumentos
6.6 Paquetes de la API de Java
6.7 Generación de números aleatorios
6.8 Ejemplo: Un juego de azar
6.9 Alcance de las declaraciones
6.10 Métodos de la clase `JApplet`
6.11 Sobrecarga de métodos
6.12 Recursividad
6.13 Ejemplo del uso de la recursividad: La serie de Fibonacci
6.14 Comparación entre recursividad e iteración
6.15 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar las operaciones de las clases

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

6.1 Introducción

La mayoría de los programas de cómputo que resuelven los problemas reales son mucho más extensos que los programas que se presentan en los primeros capítulos de este libro. La experiencia ha demostrado que la mejor manera de desarrollar y mantener un programa extenso es construirlo a partir de pequeñas piezas sencillas, o *módulos*. A esta técnica se le llama *divide y vencerás*. En este capítulo describiremos cómo declarar y utilizar métodos para facilitar el diseño, la implementación, operación y mantenimiento de programas extensos.

6.2 Módulos de programas en Java

Hay dos tipos de módulos en Java: *métodos* y *clases*. Para escribir programas en Java, se combinan nuevos métodos y clases escritas por el programador, con los métodos y clases "preempaquetados", que están disponibles en la *Interfaz de Programación de Aplicaciones de Java* (también conocida como la *API de Java* o *biblioteca de clases de Java*) y en diversas bibliotecas de clases. La API de Java proporciona una vasta colección de clases que contienen métodos para realizar cálculos matemáticos, manipulaciones de cadenas, manipulaciones de caracteres, operaciones de entrada/salida, comprobación de errores y muchas otras operaciones útiles. Este conjunto de clases facilita la escritura de programas, ya que la API de Java proporciona muchas de las herramientas que necesitan los programadores. Las clases de la API de Java forman parte del Kit de desarrollo de software para Java 2 (J2SDK), el cual contiene miles de clases preempaquetadas.



Buena práctica de programación 6.1

Procure familiarizarse con la vasta colección de clases y métodos que proporciona la API de Java (`java.sun.com/j2se/1.4.1/docs/api/index.html`) y con las extensas colecciones de clases disponibles en las diversas bibliotecas de clases.



Observación de ingeniería de software 6.1

Evite reinventar la rueda. Cuando sea posible, utilice clases y métodos de la API de Java, en vez de escribir nuevas clases y métodos. Esto reduce el tiempo de desarrollo y evita que se introduzcan errores de programación.



Tip de rendimiento 6.1

No trate de reescribir las clases y métodos existentes de la API de Java, para tratar de hacerlos más eficientes. Por lo general no podrá mejorar su rendimiento.

Los métodos (también conocidos como *funciones* o *procedimientos* en otros lenguajes de programación) permiten al programador dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas; también conocidas como *métodos declarados por el programador*. Las instrucciones que implementan los métodos se escriben sólo una vez, y están ocultas de otros métodos.

Existen diversas razones para crear un programa en módulos por medio de los métodos. Una razón es que la metodología "divide y vencerás" hace que el desarrollo de programas sea más fácil de administrar. Otra razón es la *reutilización de software* (usar los métodos existentes como bloques de construcción para crear nuevos programas). A menudo se pueden crear programas a partir de métodos estandarizados, en vez de tener que crear código personalizado. Por ejemplo, en los programas anteriores no tuvimos que definir cómo convertir cadenas en enteros y números de punto flotante; Java proporciona estas herramientas en la clase `Integer` (el método estático `parseInt`) y en la clase `Double` (el método estático `parseDouble`), respectivamente. Una tercera razón es para evitar repetir código dentro del programa. El proceso de empaquetar un código en forma de método permite que un programa ejecute ese código desde varias ubicaciones, con sólo llamar a ese método. Además, los métodos facilitan la depuración y el mantenimiento de los programas.



Observación de ingeniería de software 6.2

Para promover la reutilización de software, cada método debe limitarse de manera que realice una sola tarea bien definida, y su nombre debe expresar esa tarea con efectividad. Estos métodos hacen que los programas sean más fáciles de escribir, depurar, mantener y modificar.



Tip para prevenir errores 6.1

Los métodos pequeños que realizan una tarea son más fáciles de probar y depurar que los métodos más grandes que realizan muchas tareas.



Observación de ingeniería de software 6.3

Si no puede elegir un nombre conciso que exprese la tarea de un método, tal vez esté tratando de realizar diversas tareas en un mismo método. Por lo general, es mejor dividirlo en varios métodos más pequeños.



Observación de ingeniería de software 6.4

Generalmente, un método debe ocupar cuando mucho una página impresa; o mejor aún, la mitad de una página impresa. Sin importar qué tan largo sea un método, éste debe realizar una tarea lo mejor posible. Los métodos pequeños fomentan la reutilización de software.

Un método se *invoca* o se *llama* (es decir, se le indica que haga su tarea designada) mediante la *llamada a un método*. Esta llamada especifica el nombre del método y proporciona información (en forma de *argumentos*) que el método llamado requiere para realizar su tarea. Una vez que se ha completado la llamada, el método devuelve un resultado al *solicitante* (o *método que hizo la llamada*) o simplemente le devuelve el control. Una analogía a esta estructura de programa es la forma jerárquica de la administración. Un jefe (el solicitante) pide a un trabajador (el *método llamado*) que realice una tarea y que le reporte (es decir, *devuelva*) los resultados después de completar la tarea. El método jefe no sabe *cómo* realiza el método trabajador sus tareas designadas. Tal vez el trabajador llame a otros métodos trabajadores, sin que lo sepa el jefe. Este "ocultamiento" de los detalles de implementación fomenta la buena ingeniería de software. En la figura 6.1 se muestra al método `jefe` comunicándose con varios métodos trabajador en forma jerárquica. El método `jefe` divide las responsabilidades entre los diversos métodos `trabajadores`. Observe que `trabajador1` actúa como "método jefe" de `trabajador4` y `trabajador5`. Las relaciones entre los métodos también pueden ser distintas de la estructura jerárquica que se muestra en esta figura.

6.3 Métodos de la clase `Math`

La clase `Math` proporciona una colección de métodos que le permitirán a usted realizar cálculos matemáticos comunes. Aquí utilizamos varios métodos de la clase `Math` para introducir el concepto de los métodos. A lo largo del libro veremos más métodos de las clases API de Java.

http://librosysolucionarios.net

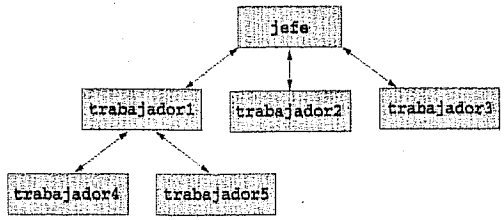


Figura 6.1 Relación jerárquica entre el método jefe y los métodos trabajadores.

Para llamar a un método se escribe su nombre seguido de un paréntesis de apertura, después su *argumento* (o una lista de argumentos separados por comas) y al final un paréntesis de cierre. Por ejemplo, un programador que desee calcular la raíz cuadrada de `900.0` podría escribir

```
Math.sqrt( 900.0 )
```

Al ejecutarse esta instrucción, llama al método estático `sqrt`[1] de `Math` para calcular la raíz cuadrada del número encerrado entre paréntesis (`900.0`). El número `900.0` es el argumento del método `sqrt`. La expresión anterior se evalúa, dando como resultado `30.0`. El método `sqrt` toma un argumento de tipo `double` y devuelve un resultado de tipo `double`. Para mostrar el valor de la llamada anterior al método en la ventana de comandos, puede escribir

```
System.out.println( Math.sqrt( 900.0 ) );
```

En esta instrucción, el valor que devuelve `sqrt` se convierte en el argumento para el método `println`.

Observación de ingeniería de software 6.5

No es necesario importar la clase `Math` para utilizar sus métodos. `Math` es parte del paquete `java.lang`, el cual se importa automáticamente por el compilador.

Error común de programación 6.1

Olvidar llamar a un método de la clase `Math`, anteponiendo al nombre del método el nombre de la clase `Math` y un punto (`.`), produce un error de sintaxis.

Los argumentos del método pueden ser constantes, variables o expresiones. Si `c = 13.0`, `d = 3.0` y `f = 4.0`, entonces la instrucción

```
System.out.println( Math.sqrt( c + d * f ) );
```

calcula e imprime la raíz cuadrada de `13.0 + 3.0 * 4.0 = 25.0`; es decir, `5.0`.

En la figura 6.2 se sintetizan varios métodos de la clase `Math`. En la figura, las variables `x` y `y` son de tipo `double`. La clase `Math` también declara dos constantes matemáticas que se utilizan comúnmente: `Math.PI` y `Math.E`. La constante `Math.PI` (3.141592653589793238462643) de la clase `Math` es la proporción entre la circunferencia de un círculo y su diámetro. La constante `Math.E` (2.718281828459045235360287) es el valor base para los logaritmos naturales (que se calculan con el método estático `log` de la clase `Math`).

6.4 Declaraciones de métodos

Los programas presentados hasta este punto consisten en la declaración de una clase, la cual contiene cuando menos la declaración de un método que llama a los métodos de la API de Java para realizar sus tareas. Ahora veremos la manera en que los programadores escriben sus propios métodos.[2]

1. Todos los métodos de la clase `Math` son `static`; por lo tanto, se llaman anteponiendo al nombre del método el nombre de la clase `Math` y un punto (`.`).
2. Hemos visto varios ejemplos de applets, en los cuales se declararon varios métodos (`init`, `start` y `paint`). Seguiremos utilizando applets para los programas con diversas declaraciones de métodos hasta que veamos los detalles sobre la declaración de clases en los capítulos 8 a 10. En ese punto, utilizaremos aplicaciones más elaboradas para muchos ejemplos.

Método	Descripción	Ejemplo

`abs( x )`	Valor absoluto de *x* (este método también tiene versiones para `float`, `int` y `long`)	`abs( 23.7 )` es `23.7`
		`abs( 0.0 )` es `0.0`
		`abs( -23.7 )` es `23.7`

`ceil( x )`	Redondea *x* al entero más pequeño que no sea menor de *x*	`ceil( 9.2 )` es `10.0` `ceil( -9.8 )` es `-9.0`
`cos( x )`	Coseno trigonométrico de *x* (*x* está en radianes)	`cos( 0.0 )` es `1.0`
`exp( x )`	método exponencial e^x	`exp( 1.0 )` es `2.71828` `exp( 2.0 )` es `7.38906`
`floor( x )`	Redondea *x* al entero más grande que no sea mayor de *x*	`floor( 9.2)` es `9.0` `floor( -9.8)` es `-10.0`
`log( x )`	Logaritmo natural de *x* (base *e*)	`log( Math.E )` es `1.0` `log( Math.E * Math.E )` es `2.0`
`max( x, y )`	El valor más grande de *x* y *y* (este método también tiene versiones para `float`, `int` y `long`)	`max( 2.3, 12.7 )` es `12.7` `max( -2.3, -12.7 )` es `-2.3`
`min( x, y )`	El valor más pequeño de *x* y *y* (este método también tiene versiones para `float`, `int` y `long`)	`min( 2.3, 12.7 )` es `2.3` `min( -2.3, -12.7 )` es `-12.7`
`pow( x, y )`	*x* elevado a la potencia *y* (x^y)	`pow( 2.0, 7.0 )` es `128.0` `pow( 9.0, 0.5 )` es `3.0`
`sin( x )`	Seno trigonométrico de *x* (*x* está en radianes)	`sin( 0.0 )` es `0.0`
`sqrt( x )`	Raíz cuadrada de *x*	`sqrt( 900.0 )` es `30.0` `sqrt( 9.0 )` es `3.0`
`tan( x )`	Tangente trigonométrica de *x* (*x* está en radianes)	`tan( 0.0 )` es `0.0`

Figura 6.2 Métodos de la clase `Math`.

Considere un applet (figura 6.3) que utiliza el método `cuadrado` para calcular los cuadrados de los enteros del 1 al 10. Cuando el applet comienza a ejecutarse, el contenedor de applets (el `appletviewer` o un navegador Web) llama al método `init` del applet (declarado en las líneas 10 a 35). En la línea 13 se declara una referencia tipo `JTextArea` llamada `areaSalida` y la inicializa con un nuevo objeto `JTextArea`. Este objeto mostrará los resultados del programa.

```
1  // Fig. 6.3: CuadradoEnteros.java
2  // Crear y utilizar un método definido por el programador.
3  import java.awt.Container;
4
5  import javax.swing.*;
6
7  public class CuadradoEnteros extends JApplet {
8
9     // configurar GUI y calcular los cuadrados de los enteros del 1 al 10
10    public void init()
11    {
12       // objeto JTextArea para mostrar resultados
13       JTextArea areaSalida = new JTextArea();
14
15       // obtener panel de contenido del applet (área visible del
         componente de la GUI)
```

Figura 6.3 Método `cuadrado` definido por el programador. (Parte 1 de 2.)

http://librosysolucionarios.net

```
16        Container contenedor = getContentPane();
17
18        // adjuntar areaSalida al contenedor
19        contenedor.add( areaSalida );
20
21        int resultado;          // guardar el resultado de la llamada
                                     al método cuadrado
22        String salida = "";     // Objeto String que contiene los resultados
23
24        // iterar 10 veces
25        for ( int contador = 1; contador <= 10; contador++ ) {
26           resultado = cuadrado( contador ); // llamada al método
27
28           // anexar resultado al objeto String salida
29           salida += "El cuadrado de " + contador + " es " + resultado + "\n";
30
31        } // fin de instrucción for
32
33        areaSalida.setText( salida );  // colocar resultados en el objeto
                                            JTextArea
34
35     } // fin del método init
36
37     // declaración del método cuadrado
38     public int cuadrado( int y )
39     {
40        return y * y;  // devolver cuadrado de y
41
42     } // fin del método cuadrado
43
44  } // fin de la clase CuadradoEnteros
```

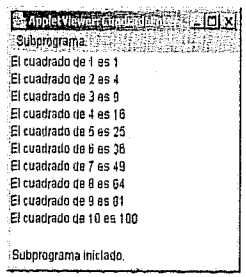


Figura 6.3 Método `cuadrado` definido por el programador. (Parte 2 de 2.)

Este programa es el primero en el que mostramos un componente de interfaz gráfica de usuario (GUI) en un applet. El área visible en pantalla para un objeto de la clase `JApplet` tiene un *panel de contenido* (un objeto de la clase `Container`, del *paquete* `java.awt`), al que se deben adjuntar los componentes de la GUI para que puedan mostrarse en tiempo de ejecución. Este objeto es proporcionado por el applet, por lo que nuestro programa no necesita crear un objeto `Container`. La clase `Container` se importó en la línea 3 para utilizarla en el applet. En la línea 16 se declara la referencia `contenedor` de tipo `Container` y le asigna el resultado de una llamada al método `getContentPane`; uno de los muchos métodos que nuestra clase `CuadradoEnteros` hereda de la clase `JApplet`. Este método devuelve una referencia al objeto del panel de contenido del applet. El programa utiliza esa referencia para adjuntar componentes de la GUI, como un objeto `JTextArea`, a la interfaz de usuario del applet.

http://librosysolucionarios.net

Cuando el applet se ejecuta, se muestran todos los componentes de la GUI adjuntos al panel de contenido del applet. Por lo tanto, en la línea 19 se adjunta el objeto `JTextArea`, al que `areaSalida` hace referencia, al panel de contenido del applet, utilizando la referencia `contenedor` para que llame al método `add` de la clase `Container`. Por el momento, podemos adjuntar sólo un componente de la GUI al panel de contenido del applet, y ese componente de la GUI ocupará toda el área de dibujo del applet en la pantalla (según esté definida por los atributos `width` y `height` del applet en su documento HTML). En la sección 6.8 veremos cómo adjuntar muchos componentes de la GUI a un applet, cambiando el *esquema* del mismo, el cual controla la manera en que el applet coloca a los componentes de la GUI en su área en la pantalla.

En la línea 21 se declara la variable `resultado` tipo `int` para guardar el resultado de cada cálculo del cuadrado. En la línea 22 se declara la referencia `salida` tipo `String` y se inicializa con una cadena vacía. Esta cadena contendrá los resultados de calcular los cuadrados de los valores del 1 al 10. En las líneas 25 a 31 se define una instrucción `for`. Durante cada una de las iteraciones de este ciclo, en la línea 26 se calcula el `cuadrado` del valor actual de la variable de control `contador`, y guarda el valor en `resultado`; En la línea 29 se concatena `resultado` al final de la cadena `salida`.

El applet llama a su método `cuadrado` en la línea 26, mediante la instrucción

```
resultado = cuadrado( contador );
```

Los paréntesis a la derecha de `cuadrado` indican la llamada a un método y encierran la lista de argumentos de ese método. En este punto, el programa hace una copia del valor de `contador` (el argumento de la llamada al método) y transfiere el control del programa a la primera línea del método `cuadrado` (declarado en las líneas 38 a 42). En la línea 38 de la declaración del método `cuadrado` se muestra que `cuadrado` espera un argumento entero, según lo especifica su lista de parámetros. El método `cuadrado` utiliza el *parámetro*[3] `y` para manipular el valor que `cuadrado` recibe como argumento. La palabra clave `int` que precede al nombre del método, indica que `cuadrado` devuelve un resultado entero. El método `cuadrado` recibe la copia del valor de `contador` en el parámetro `y`. Después, `cuadrado` calcula `y * y` (línea 40). El método `cuadrado` utiliza una instrucción `return` para devolver (es decir, regresar) el resultado del cálculo a la instrucción en el método `init` que llamó a `cuadrado`. Observe que toda la declaración del método (líneas 38 a 42) aparece entre las llaves de la clase `CuadradoEnteros`. Todos los métodos deben declararse dentro de la declaración de una clase.



Buena práctica de programación 6.2

Coloque una línea en blanco entre las declaraciones de los métodos, para separarlos y mejorar la legibilidad del programa.



Error común de programación 6.2

Declarar un método fuera de las llaves de la declaración de una clase es un error de sintaxis.

En el método `init`, el valor de retorno se asigna a la variable `resultado`. En la línea 29 se concatena `"El cuadrado de"`, el valor de `contador`, `"es"`, el valor de `resultado` y un carácter de nueva línea, al final de la cadena `salida`. Este proceso se repite para cada iteración de la instrucción `for`. En la línea 33 se utiliza la referencia `areaSalida` para llamar al método `setText` del objeto `JTextArea`, el cual establece el texto (`salida`) a mostrar en el área de texto.

Declaramos las referencias `areaSalida`, `contenedor` y `salida`, junto con la variable `resultado`, como variables locales en `init` ya que se utilizan sólo en ese método. Las variables deben declararse como campos solamente si se requiere su uso en más de un método de la clase, o si el programa debe guardar sus valores entre llamadas a los métodos de la clase.

Observe que el método `init` llama al método `cuadrado` directamente, sin calificar el nombre del método con el nombre de una clase y un punto (`.`), o con el nombre de una referencia y un punto. Cada método de una clase tiene la habilidad de llamar a los demás métodos de esa clase en forma directa.[4]

3. A los parámetros se les conoce también como *parámetros formales*.
4. Hay una excepción a esta regla: los métodos `static` de una clase pueden llamar solamente a otros métodos `static` de la clase en forma directa. En el capítulo 8 hablaremos detalladamente sobre los métodos `static`.

Formato general de la declaración de un método

La primera línea de la declaración de un método se conoce como *encabezado del método*. Después del encabezado del método van las *declaraciones* e *instrucciones* entre llaves, las cuales forman el *cuerpo del método*, que es un bloque. Pueden declararse variables en cualquier bloque, y los bloques pueden anidarse. Un método no puede declararse dentro de otro método.

El formato básico de la declaración de un método es

```
tipo-de-valor-de-retorno  nombre-del-método( parámetro1, parámetro2, ..., parámetroN )
{
    declaraciones e instrucciones
}
```

El *nombre-del-método* es cualquier identificador válido. El *tipo-de-valor-de-retorno* es el tipo del resultado devuelto por el método al solicitante. El *tipo-de-valor-de-retorno* `void` indica que un método no devuelve un valor. Los métodos pueden devolver como máximo un valor.



Error común de programación 6.3

Omitir el tipo de valor de retorno *en la declaración de un método es un error de sintaxis.*



Error común de programación 6.4

Olvidar devolver un valor de un método que debe devolver un valor es un error de sintaxis. Si se especifica un tipo de valor de retorno que no sea `void`*, el método debe contener una instrucción* `return` *que devuelva un valor del tipo de valor de retorno del método. De manera similar, devolver un valor de un método cuyo tipo de valor de retorno se haya declarado como* `void` *es un error de sintaxis.*

Los *parámetros* se declaran en una lista separada por comas, encerrada entre paréntesis, que declara el tipo y nombre de cada uno de los parámetros. Debe haber un argumento en la llamada al método para cada parámetro en la declaración del método. Además, cada argumento debe ser compatible con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores de 7.35, 22 o –0.03546, pero no `"hello"` (porque una cadena no puede convertirse implícitamente en una variable `double`). Si un método no acepta argumentos, la lista de parámetros está vacía (es decir, el nombre del método va seguido de un conjunto vacío de parámetros).



Error común de programación 6.5

Declarar los parámetros de un método del mismo tipo como `float x, y` *(que es una sintaxis de declaración de variables) en vez de* `float x, float y` *es un error de sintaxis, ya que se requiere un tipo para cada parámetro de la lista de parámetros.*



Error común de programación 6.6

Colocar un punto y coma después del paréntesis derecho que encierra a la lista de parámetros de la declaración de un método, es un error de sintaxis.



Error común de programación 6.7

Declarar nuevamente el parámetro de un método como variable local en el cuerpo del método, es un error de sintaxis.



Error común de programación 6.8

Pasar un argumento que no es compatible con el tipo del parámetro correspondiente en la lista de parámetros del método, es un error de sintaxis (por ejemplo, pasar un `String` *cuando se espera un* `int`*).*



Buena práctica de programación 6.3

Elegir nombres significativos para los métodos y los parámetros hace que los programas sean más legibles, y ayuda a evitar un uso excesivo de comentarios.



Observación de ingeniería de software 6.6

El encabezado del método y las llamadas al método deben coincidir en cuanto al número, tipo y orden de los parámetros y argumentos.

http://librosysolucionarios.net



Observación de ingeniería de software 6.7

Un método que tenga muchos parámetros tal vez esté realizando demasiadas tareas. Considere dividir el método en métodos más pequeños que realicen las tareas por separado. El encabezado del método debe caber en una línea, si es posible.

Hay tres maneras de regresar el control a la instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de finalización del método, o cuando se ejecuta la instrucción

```
return;
```

si el método devuelve un resultado, la instrucción

```
return expresión;
```

evalúa la *expresión* y después devuelve el valor resultante al solicitante. Cuando se ejecuta una instrucción `return`, el control regresa inmediatamente a la instrucción que llamó al método.

Notas sobre la sintaxis para llamar a un método

La figura 6.3 contiene las declaraciones de dos métodos: `init` (líneas 10 a 35) y `cuadrado` (líneas 38 a 42). Recuerde que en la sección 3.5 vimos que el contenedor de applets llama al método `init` para inicializar el applet. En este ejemplo, el método `init` llama repetidamente al método `cuadrado` para realizar un cálculo; después coloca los resultados en el objeto `JTextArea` que se adjunta al panel de contenido del applet. Cuando el applet aparece en la pantalla, los resultados se muestran en el área de texto.

Observe la sintaxis utilizada para llamar al método `cuadrado`; simplemente utilizamos el nombre del método, seguido por los argumentos para el método entre paréntesis. Los métodos de la declaración de una clase pueden llamar a los demás métodos en la misma declaración de clase utilizando esta sintaxis.[5] Los métodos en la misma declaración de clase incluyen tanto a los métodos declarados en esa clase, como a los métodos heredados en esa clase (es decir, los métodos que la clase actual extiende mediante la instrucción `extends JApplet` en la figura 6.3; vea la línea 7]). Ahora hemos visto tres maneras de llamar a un método:

1. El nombre del método por sí solo; como `cuadrado( contador )` en la línea 26 de la figura 6.3.
2. La referencia a un objeto, seguido de un punto (.) y el nombre del método; como `g.drawLine( 15, 10, 210, 10)` en la línea 17 de la figura 3.11.
3. El nombre de la clase calificando al nombre de un método; como `Integer.parseInt( primerNumero )` en la línea 28 de la figura 2.9.

La última sintaxis es sólo para métodos de la clase `static` (los cuales describiremos detalladamente en el capítulo 8).

Método `maximo` definido por el programador

El applet de la figura 6.4 utiliza un método llamado `maximo`, definido por el programador para determinar y devolver el mayor de tres valores de punto flotante.

En las líneas 13 a 18 del método `init` se utilizan cuadros de diálogo de entrada para obtener del usuario tres valores de punto flotante. En las líneas 21 a 23 se utiliza el método `Double.parseDouble` para convertir las cadenas introducidas por el usuario en valores `double`. En la línea 25 se llama al método `maximo` (declarado en las líneas 44-48) para determinar el valor `double` más grande de los tres valores `double` que se pasan como argumentos al método. El método `maximo` devuelve el resultado al método `init`, utilizando una instrucción `return`. El programa asigna el resultado a la variable `max` en la línea 25. En las líneas 31 y 32 se utiliza la concatenación de objetos `String` para formar una cadena, que contiene los tres valores `double` introducidos por el usuario y el valor `max`, y después se coloca el resultado en el área de texto `areaSalida`.

Considere la declaración del método `maximo` (líneas 44 a 48). En la línea 44 se indica que el método devuelve un valor `double` de punto flotante, que el nombre del método es `maximo` y que el método requiere tres

5. Hay una excepción a esta regla, la cual se describe en el capítulo 8.

parámetros `double` (`x`, `y` y `z`) para realizar su tarea. En la línea 46 se devuelve el valor máximo de los tres valores de punto flotante, utilizando dos llamadas al método `Math.max`. Primero, la instrucción llama a `Math.max` con los valores de las variables `y` y `z` para determinar el mayor de los dos valores. Después, la instrucción pasa el valor de la variable `x` y el resultado de la primera llamada a `Math.max` como argumentos para el método `Math.max`. Finalmente, la instrucción devuelve el resultado de la segunda llamada a `Math.max` a la línea 25 (el punto en el que el método `init` llamó al método `maximo`).

```
1  // Fig. 6.4: PruebaMaximo.java
2  // Encontrar el valor máximo de tres números de punto flotante.
3  import java.awt.Container;
4
5  import javax.swing.*;
6
7  public class PruebaMaximo extends JApplet {
8
9     // inicializar el applet, obtener la entrada del usuario y crear GUI
10    public void init()
11    {
12       // obtener entrada del usuario
13       String s1 = JOptionPane.showInputDialog(
14          "Escriba el primer valor de punto flotante" );
15       String s2 = JOptionPane.showInputDialog(
16          "Escriba el segundo valor de punto flotante" );
17       String s3 = JOptionPane.showInputDialog(
18          "Escriba el tercer valor de punto flotante" );
19
20       // convertir la entrada del usuario en valores double
21       double numero1 = Double.parseDouble( s1 );
22       double numero2 = Double.parseDouble( s2 );
23       double numero3 = Double.parseDouble( s3 );
24
25       double max = maximo( numero1, numero2, numero3 ); // llamada al método
26
27       // crear objeto JTextArea para mostrar los resultados
28       JTextArea areaSalida = new JTextArea();
29
30       // mostrar los números y el valor máximo
31       areaSalida.setText( "numero1: " + numero1 + "\nnumero2: " +
32          numero2 + "\nnumero3: " + numero3 + "\nel valor máximo es: " + max );
33
34       // obtener el área visible del applet para componentes GUI
35       Container contenedor = getContentPane();
36
37       // adjuntar areaSalida al contenedor
38       contenedor.add( areaSalida );
39
40    } // fin del método init
41
42    // el método maximo utiliza el método max de la clase Math para
43    // determinar el valor máximo
44    public double maximo( double x, double y, double z )
45    {
46       return Math.max( x, Math.max( y, z ) );
47
48    } // fin del método maximo
49
50 } // fin de la clase PruebaMaximo
```

Figura 6.4 Método `maximo` definido por el programador. (Parte 1 de 2.)

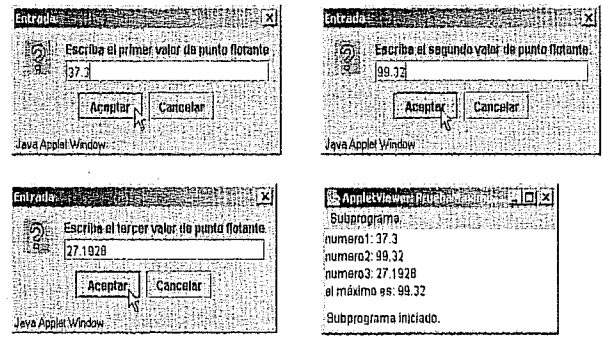


Figura 6.4 Método `maximo` definido por el programador. (Parte 2 de 2.)

6.5 Promoción de argumentos

Otra característica importante de las llamadas a métodos es la *promoción de argumentos* (o *coerción de argumentos*); es decir, forzar a que se pasen argumentos del tipo apropiado a un método. Por ejemplo, un programa puede llamar al método `sqrt` de `Math` con un argumento entero, inclusive cuando el método espera recibir un argumento `double`. Por ejemplo, la instrucción

```
System.out.println( Math.sqrt( 4 ) );
```

evalúa correctamente `Math.sqrt( 4 )` e imprime el valor `2`. La lista de parámetros de la declaración del método hace que Java convierta el valor entero `4` en el valor `double` `4.0` antes de pasar el valor a `sqrt`. En algunos casos, intentar estas conversiones ocasiona errores de compilación si no se satisfacen las *reglas de promoción* de Java. Estas reglas especifican cómo convertir de un tipo a otro sin perder datos. En el ejemplo anterior con `sqrt`, un `int` se convierte en `double` sin cambiar su valor. Sin embargo, al convertir un `double` en `int` se trunca la parte fraccionaria del valor `double`; por lo tanto, se pierde parte del valor. La conversión de tipos enteros grandes a tipos enteros pequeños (por ejemplo, de `long` a `int`) puede producir también valores modificados.

Las reglas de promoción se aplican a las expresiones que contienen valores de dos o más tipos primitivos (también conocidos como *expresiones de tipo mixto*) y a los valores de tipos primitivos que se pasan como argumentos a los métodos. El tipo de cada valor en una expresión de tipo mixto se promueve al tipo "más alto" en la expresión. (En realidad, la expresión utiliza una copia temporal de cada valor; los valores originales permanecen sin cambios.) La figura 6.5 muestra los tipos primitivos y los tipos a los que puede promoverse cada uno de ellos. Observe que todas las promociones válidas para un tipo dado son los tipos "de más arriba" en la tabla. Por ejemplo, un `int` puede promoverse a los tipos `long`, `float` y `double` que están más arriba.

Convertir valores a tipos que estén más abajo en la tabla de la figura 6.5 puede producir valores distintos. Por lo tanto, en casos en los que la información podría perderse debido a la conversión, el compilador de Java requiere al programador que utilice un operador de conversión de tipos (vea la sección 4.9) para forzar explícitamente a que ocurra la conversión. Nuestro método `cuadrado` (de la figura 6.3) utiliza un parámetro entero. Para llamar a `cuadrado` con una variable `double` llamada `contador`, tendríamos que escribir la llamada al método como `cuadrado ( (int) contador )`. Esta llamada al método convierte explícitamente el valor de `y` en un entero, para usarlo en el método `cuadrado`. Por lo tanto, si el valor de `y` es `4.5`, el método `cuadrado` devuelve `16`, no `20.25`.

http://librosysolucionarios.net

Tipo	Promociones válidas
`double`	Ninguna
`float`	`double`
`long`	`float` o `double`
`int`	`long`, `float` o `double`
`char`	`int`, `long`, `float` o `double`
`short`	`int`, `long`, `float` o `double`
`byte`	`short`, `int`, `long`, `float` o `double`
`boolean`	Ninguno (los valores `boolean` no se consideran como números en Java)

Figura 6.5 Promociones permitidas para los tipos primitivos.



Error común de programación 6.9

Convertir un valor de un tipo primitivo en otro tipo primitivo puede cambiar ese valor, si el nuevo tipo no es una promoción válida. Por ejemplo, convertir un valor de punto flotante en un valor entero puede introducir errores de truncamiento (pérdida de la parte fraccionaria) en el resultado.

6.6 Paquetes de la API de Java

Como hemos visto, Java contiene muchas clases predefinidas que se agrupan en categorías de clases relacionadas, llamadas *paquetes*. En conjunto, nos referimos a estos paquetes como la *API de Java (Interfaz de programación de aplicaciones de Java)*, o *biblioteca de clases de Java*.

A lo largo del texto, las declaraciones `import` especifican las clases requeridas para compilar un programa en Java. Por ejemplo, un programa incluye la declaración

```
import javax.swing.JApplet;
```

para especificar que el programa utiliza la clase `JApplet` del paquete `javax.swing`. Esto permite a los programadores utilizar el nombre de la clase `JApplet`, en vez de tener que usar el nombre completo calificado de la clase, `javax.swing.JApplet`, en el código. Uno de los puntos más fuertes de Java es el extenso número de clases en los paquetes de la API de Java, que los programadores pueden reutilizar. Utilizaremos una gran cantidad de esas clases en este libro. La figura 6.6 proporciona una breve descripción de un subconjunto de los paquetes en la API de Java. Utilizaremos clases de estos paquetes y de otros más a lo largo de este libro. Le proporcionamos este listado para introducirlo a la variedad de componentes reutilizables que hay disponibles en la API de Java. Mientras esté aprendiendo este lenguaje, invierta una parte de su tiempo explorando las descripciones de los paquetes y las clases en la documentación para la API de Java (`java.sun.com/j2se/1.4.1/docs/api`).

El conjunto de paquetes disponibles en el Kit de desarrollo de software para Java 2 (J2SDK) es bastante extenso. Además de los paquetes sintetizados en la figura 6.6, el J2SDK incluye paquetes para gráficos complejos, interfaces gráficas de usuario avanzadas, impresión, redes avanzadas, seguridad, procesamiento de bases de datos, multimedia, accesibilidad (para personas con discapacidades) y muchas otras funciones. Para una visión general de los paquetes en el J2SDK versión 1.4.1, visite:

```
java.sun.com/j2se/1.4.1/docs/api/overview-summary.html
```

Además, muchos otros paquetes están disponibles para descargarse en `java.sun.com`.

6.7 Generación de números aleatorios

Ahora analizaremos de manera breve una parte divertida de la programación: simulación y juegos. En ésta y en la siguiente sección desarrollaremos un programa de juego bien estructurado con varios métodos. El programa

http://librosysolucionarios.net

Paquete	Descripción
`java.applet`	El *Paquete Applet de Java* contiene la clase `Applet` y varias interfaces que permiten la interacción entre applets y navegadores, y la reproducción de clips de audio. En Java 2, la clase `javax.swing.JApplet` se utiliza para definir un applet que utiliza los componentes de la GUI de Swing.
`java.awt`	El *Paquete Abstract Window Toolkit de Java* contiene las clases e interfaces requeridas para crear y manipular GUIs en Java 1.0 y 1.1. En Java 2 se utilizan los componentes de la GUI de Swing incluidos en el paquete `javax.swing`.
`java.awt.event`	El *Paquete Abstract Window Toolkit Event de Java* contiene clases e interfaces que habilitan el manejo de eventos para componentes de la GUI en los paquetes `java.awt` y `javax.swing`.
`java.io`	El *Paquete de entrada/salida de Java* contiene clases que permiten a los programas recibir datos de entrada y mostrar datos de salida (vea el capítulo 17).
`java.lang`	El *Paquete del Lenguaje Java* contiene clases e interfaces (descritas a lo largo de este texto) requeridas por muchos programas de Java. Este paquete es importado por el compilador en todos los programas.
`java.net`	*El Paquete de red de Java* contiene clases que permiten a los programas comunicarse mediante redes (vea el capítulo 18).
`java.text`	El *Paquete de texto de Java* contiene clases e interfaces que permiten a un programa de Java manipular números, fechas, caracteres y cadenas. El paquete proporciona muchas de las herramientas de internacionalización de Java que permiten la personalización de un programa con respecto a una configuración regional específica (por ejemplo, un applet puede mostrar cadenas en distintos lenguajes, con base en el país del usuario).
`java.util`	El *Paquete de utilerías de Java* contiene clases e interfaces utilitarias, como manipulaciones de fecha y hora, herramientas para procesar números aleatorios con la clase `Random`, almacenar y procesar grandes cantidades de datos y descomponer cadenas en piezas más pequeñas llamadas *tokens*, con la clase `StringTokenizer` (vea los capítulos 20, 21 y 22).
`javax.swing`	El *Paquete de componentes GUI Swing de Java* contiene clases e interfaces para los componentes de la GUI Swing de Java, los cuales ofrecen soporte para GUIs portables.
`javax.swing.event`	El *Paquete Swing Event de Java* contiene clases e interfaces que permiten el manejo de eventos para los componentes de la GUI en el paquete `javax.swing`.

Figura 6.6 Paquetes de la API de Java (un subconjunto).

utiliza la mayoría de las instrucciones de control presentadas hasta este punto en el libro, e introduce varios conceptos nuevos.

Hay algo en el ambiente de un casino de apuestas que anima a las personas: desde las elegantes mesas de caoba y fieltro para tirar dados, hasta las máquinas tragamonedas. Es el *elemento de azar*, la posibilidad de que la suerte convierta un bolsillo lleno de dinero en una montaña de riquezas. El elemento de azar puede introducirse mediante el método *`random`* de la clase `Math`.[6]

Considere la siguiente instrucción:

```
double valorAleatorio = Math.random();
```

El método `random` de `Math` genera un valor `double` aleatorio en el rango de 0.0 hasta, pero sin incluir, 1.0. Si el método `random` verdaderamente produce valores aleatorios, entonces cualquier valor desde 0.0 hasta, pero sin incluir, 1.0 debería tener una *oportunidad* (o *probabilidad*) igual de ser elegido cada vez que se llame a `random`. Los valores devueltos por `random` son en realidad *números seudoaleatorios* (una secuencia de va-

[6] Otra manera proporciona la clase `Random` en el paquete `java.util` para producir valores aleatorios. Un objeto de esta clase puede producir valores aleatorios `int`, `long`, `float` o `double`.

lores producidos por un cálculo matemático complejo). Ese cálculo utiliza la hora actual del día para *sembrar* el generador de números aleatorios de tal forma que la ejecución de un programa produzca una secuencia distinta de valores aleatorios.

El rango de valores producidos directamente por el método **random** es a menudo distinto del rango de valores requeridos en una aplicación de Java particular. Por ejemplo, un programa que simula el tiro de un dado de seis lados requeriría enteros aleatorios en el rango de 1 a 6. Un programa que adivine en forma aleatoria el siguiente tipo de nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte en un videojuego requeriría números aleatorios en el rango de 1 a 4.

Para demostrar el método **random**, desarrollaremos un programa que simula 20 tiros de un dado de seis lados, y que muestra el valor de cada tiro. Utilizaremos el operador de multiplicación (*) en conjunto con el método **random**, como se muestra a continuación, para producir enteros en el rango de 0 a 5.

```
( int ) (Math.random() * 6 )
```

A esta manipulación se le conoce como *escalar* el rango de valores producidos por el método **random** de **Math**. El número 6 en la expresión se conoce como el *factor de escala*. El operador de conversión a enteros trunca la parte de punto flotante (la parte después del punto decimal) de cada valor producido por la expresión. Después vamos a *desplazar* el rango de números producidos agregando un *valor de desplazamiento* (en este caso, 1) a nuestro resultado anterior, como en

```
1 + ( int ) ( Math.random() * 6 )
```

El valor de desplazamiento especifica el primer valor en el conjunto deseado de enteros aleatorios. La figura 6.7 confirma que los resultados del cálculo anterior sean enteros en el rango de 1 a 6. La línea 16 en el método **main** de la aplicación se ejecuta 20 veces en un ciclo, y los resultados se muestran en un cuadro de diálogo de mensaje.

```
1  // Fig. 6.7: EnterosAleatorios.java
2  // Enteros aleatorios con desplazamiento y escala.
3  import javax.swing.JOptionPane;
4
5  public class EnterosAleatorios {
6
7     public static void main( String args[] )
8     {
9        int valor;
10       String salida = "";
11
12       // iterar 20 veces
13       for ( int contador = 1; contador <= 20; contador++ ) {
14
15          // elegir entero aleatorio entre 1 y 6
16          valor = 1 + ( int ) ( Math.random() * 6 );
17
18          salida += valor + "   ";  // anexar valor a salida
19
20          // si el contador es divisible entre 5, anexar nueva línea a
               String salida
21          if ( contador % 5 == 0 )
22             salida += "\n";
23
24       } // fin de instrucción for
25
26       JOptionPane.showMessageDialog( null, salida,
```

Figura 6.7 Enteros aleatorios con desplazamiento y escala. (Parte 1 de 2.)


```
27          "20 números aleatorios de 1 a 6",
28          JOptionPane.INFORMATION_MESSAGE );
29
30       System.exit( 0 );  // terminar la aplicación
31
32    } // fin de main
33
34 } // fin de la clase EnterosAleatorios
```



Figura 6.7 Enteros aleatorios con desplazamiento y escala. (Parte 2 de 2.)

Para mostrar que estos números ocurren con una probabilidad aproximadamente igual, vamos a simular 6000 tiros de un dado con la aplicación de la figura 6.8. Cada entero del 1 al 6 debe aparecer aproximadamente 1000 veces.

```
1  // Fig. 6.8: TirarDado.java
2  // Tirar un dado de seis lados 6000 veces.
3  import javax.swing.*;
4
5  public class TirarDado {
6
7     public static void main( String args[] )
8     {
9        int frecuencia1 = 0, frecuencia2 = 0, frecuencia3 = 0,
10          frecuencia4 = 0, frecuencia5 = 0, frecuencia6 = 0, cara;
11
12       // sintetizar resultados
13       for ( int tiro = 1; tiro <= 6000; tiro++ ) {
14          cara = 1 + ( int ) ( Math.random() * 6 );
15
16          // determinar el valor de tiro e incrementar el contador apropiado
17          switch ( cara ) {
18
19             case 1:
20                ++frecuencia1;
21                break;
22
23             case 2:
24                ++frecuencia2;
25                break;
26
27             case 3:
28                ++frecuencia3;
29                break;
30
```

Figura 6.8 Tirar un dado de seis lados 6000 veces. (Parte 1 de 2.)

http://librosysolucionarios.net

```
31              case 4:
32                 ++frecuencia4;
33                 break;
34
35              case 5:
36                 ++frecuencia5;
37                 break;
38
39              case 6:
40                 ++frecuencia6;
41                 break;
42
43           } // fin de instrucción switch
44
45        } // fin de instrucción for
46
47        JTextArea areaSalida = new JTextArea();
48
49        areaSalida.setText( "Cara\tFrecuencia" + "\n1\t" + frecuencia1 +
50           "\n2\t" + frecuencia2 + "\n3\t" + frecuencia3 +
51           "\n4\t" + frecuencia4 + "\n5\t" + frecuencia5 +
52           "\n6\t" + frecuencia6 );
53
54        JOptionPane.showMessageDialog( null, areaSalida,
55           "Tirar un dado 6000 veces", JOptionPane.INFORMATION_MESSAGE );
56
57        System.exit( 0 );  // terminar la aplicación
58
59     } // fin de main
60
61  } // fin de la clase TirarDado
```

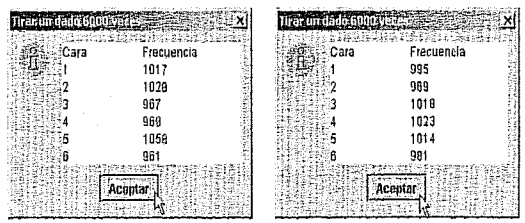


Figura 6.8 Tirar un dado de seis lados 6000 veces. (Parte 2 de 2.)

Como lo muestran las dos salidas de ejemplo, al escalar y desplazar los valores producidos por el método **random**, el programa puede simular en forma realista el tiro de un dado de seis lados. Se utilizan instrucciones de control anidadas en el programa para determinar el número de veces que se tiró cada uno de los lados del dado. La instrucción **for** de las líneas 13 a 45 itera 6000 veces. Durante cada iteración, en la línea 14 se produce un valor del 1 al 6. El valor de **cara** que se eligió al azar se utiliza como la expresión controladora de la instrucción **switch** anidada (líneas 17 a 43). Con base en el valor de **cara**, la instrucción **switch** incrementa una de las seis variables contadores durante cada iteración del ciclo.[7] Observe que la instrucción **switch** no tiene caso **default**, ya que tenemos un caso para cada posible valor del dado. Ejecute el programa varias veces y observe los resultados. Cada vez que ejecute este programa, observe que produce resultados distintos.

7. Cuando estudiemos los arreglos, en el capítulo 7, veremos cómo reemplazar toda la instrucción **switch** de este programa con una sola instrucción.

http://librosysolucionarios.net

Anteriormente demostramos la instrucción

```
cara = 1 + ( int ) ( Math.random() * 6 );
```

la cual simula el tiro de un dado de seis lados. Esta instrucción siempre asigna a la variable **cara** un entero en el rango $1 \leq$ **cara** ≤ 6. La anchura de este rango (es decir, el número de enteros consecutivos en el rango) es de 6, y el número inicial en el rango es 1. Haciendo referencia a la instrucción anterior, podemos ver que la anchura del rango se determina en base al número 6 utilizado para escalar **random** con el operador de multiplicación, y el número inicial del rango es igual al número 1 sumado a **(int) (Math.random() * 6)**. Podemos generalizar este resultado como

numero = *valorDesplazamiento* + **(int) (Math.random() *** *factorEscala* **);**

en donde *valorDesplazamiento* es igual al primer número en el rango deseado de enteros consecutivos y *factorEscala* es igual a la anchura de ese rango. En los ejercicios veremos que es posible seleccionar enteros al azar, a partir de conjuntos de valores que no sean rangos de enteros consecutivos.

6.8 Ejemplo: Un juego de azar

Uno de los juegos más populares de azar es el juego de dados conocido como "craps", el cual se juega en casinos y callejones en todo el mundo. Las reglas del juego son simples:

> *Un jugador tira dos dados. Cada dado tiene seis caras, las cuales contienen uno, dos, tres, cuatro, cinco y seis puntos negros, respectivamente. Una vez que los dados dejan de moverse, se calcula la suma de los puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado "craps"), el jugador pierde (es decir, la "casa" gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esa suma se convierte en el "punto" del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez "su punto" (es decir, que tire ese mismo valor de punto). El jugador pierde si tira un 7 antes de llegar a su punto.*

El applet de la figura 6.9 simula el juego de craps, utilizando varios métodos para definir la lógica del juego.

```
1  // Fig. 6.9: Craps.java
2  // Craps.
3  import java.awt.*;              // Container, FlowLayout
4  import java.awt.event.*;        // ActionEvent, ActionListener
5
6  import javax.swing.*;           // JApplet, JButton, JLabel, JTextField
7
8  public class Craps extends JApplet implements ActionListener {
9
10    // variables constantes para el estado del juego
11    final int GANADO = 0, PERDIDOS = 1, CONTINUAR = 2;
12
13    boolean primerTiro = true;    // verdadero si es el primer tiro del dado
14    int sumaDeDados = 0;          // suma de los dados
15    int miPunto = 0;              // punto si no gana o pierde
                                    en el primer tiro
16    int estadoJuego = CONTINUAR; // el juego no se acaba todavía
17
18    // componentes de la interfaz gráfica de usuario
19    JLabel dado1Etiqueta, dado2Etiqueta, sumaEtiqueta, puntoEtiqueta;
20    JTextField dado1Campo, dado2Campo, sumaCampo, puntoCampo;
21    JButton tirarBoton;
22
23    // configurar componentes de la GUI
```

Figura 6.9 Simulación de craps. (Parte 1 de 4.)

```
24    public void init()
25    {
26       // obtener panel de contenido y cambiar su esquema a FlowLayout
27       Container contenedor = getContentPane();
28       contenedor.setLayout( new FlowLayout() );
29
30       // crear etiqueta y campo de texto para el dado 1
31       dado1Etiqueta = new JLabel( "Dado 1" );
32       contenedor.add( dado1Etiqueta );
33       dado1Campo = new JTextField( 10 );
34       dado1Campo.setEditable( false );
35       contenedor.add( dado1Campo );
36
37       // crear etiqueta y campo de texto para el dado 2
38       dado2Etiqueta = new JLabel( "Dado 2" );
39       contenedor.add( dado2Etiqueta );
40       dado2Campo = new JTextField( 10 );
41       dado2Campo.setEditable( false );
42       contenedor.add( dado2Campo );
43
44       // crear etiqueta y campo de texto para suma
45       sumaEtiqueta = new JLabel( "La suma es" );
46       contenedor.add( sumaEtiqueta );
47       sumaCampo = new JTextField( 10 );
48       sumaCampo.setEditable( false );
49       contenedor.add( sumaCampo );
50
51       // crear etiqueta y campo de texto para punto
52       puntoEtiqueta = new JLabel( "El punto es" );
53       contenedor.add( puntoEtiqueta );
54       puntoCampo = new JTextField( 10 );
55       puntoCampo.setEditable( false );
56       contenedor.add( puntoCampo );
57
58       // crear el botón que el usuario oprime para tirar los dados
59       tirarBoton = new JButton( "Tirar dados" );
60       tirarBoton.addActionListener( this );
61       contenedor.add( tirarBoton );
62
63    } // fin del método init
64
65    // procesar un tiro de los dados
66    public void actionPerformed( ActionEvent actionEvent )
67    {
68       sumaDeDados = tirarDados();  // tirar los dados
69
70       // primer tiro de los dados
71       if ( primerTiro ) {
72
73          switch ( sumaDeDados ) {
74
75             // gana en primer tiro
76             case 7:
77             case 11:
78                estadoJuego = GANADO;
```

Figura 6.9 Simulación de craps. (Parte 2 de 4.)


```
79                puntoCampo.setText( "" );  // borrar campo punto
80                break;
81
82             // pierde en primer tiro
83             case 2:
84             case 3:
85             case 12:
86                estadoJuego = PERDIDOS;
87                puntoCampo.setText( "" );  // borrar campo punto
88                break;
89
90             // recuerda el punto
91             default:
92                estadoJuego = CONTINUAR;
93                miPunto = sumaDeDados;
94                puntoCampo.setText( Integer.toString( miPunto ) );
95                primerTiro = false;
96                break;
97
98          } // fin de instrucción switch
99
100      } // fin de la parte if de la instrucción if...else
101
102      else { // tiro subsecuente de los dados
103
104         // determinar estado del juego
105         if ( sumaDeDados == miPunto )  // gana llegando al punto
106            estadoJuego = GANADO;
107         else
108           if ( sumaDeDados == 7 )      // pierde tirando un 7
109              estadoJuego = PERDIDOS;
110
111      } // fin de la parte else de la instrucción if...else
112
113      mostrarMensaje();  // mostrar un mensaje indicando el estado del juego
114
115   } // fin del método actionPerformed
116
117   // tirar dados, calcular suma y mostrar resultados
118   public int tirarDados()
119   {
120      // elegir valores aleatorios para los dados
121      int dado1 = 1 + ( int ) ( Math.random() * 6 );
122      int dado2 = 1 + ( int ) ( Math.random() * 6 );
123
124      int suma = dado1 + dado2;   // sumar valores de los dados
125
126      // mostrar resultados en campos de texto
127      dado1Campo.setText( Integer.toString( dado1 ) );
128      dado2Campo.setText( Integer.toString( dado2 ) );
129      sumaCampo.setText( Integer.toString( suma ) );
130
131      return suma;  // devolver suma de los dados
132
133   } // fin del método tirarDados
```

http://librosysolucionarios.net

Figura 6.9 Simulación de craps. (Parte 3 de 4.)

```
134
135    // determinar estado del juego; mostrar el mensaje apropiado
          en la barra de estado
136    public void mostrarMensaje()
137    {
138       // el juego debe continuar
139       if ( estadoJuego == CONTINUAR )
140          showStatus( "Tirar de nuevo." );
141
142       else { // juego ganado o perdido
143
144          if ( estadoJuego == GANADO )
145             showStatus( "El jugador gana. Haga clic en Tirar dados para
                jugar de nuevo." );
146          else
147             showStatus( "El jugador pierde. Haga clic en Tirar dados para
                jugar de nuevo." );
148
149          primerTiro = true;  // el siguiente tiro es el primero de un
                                                          nuevo juego
150
151       } // fin de la parte else de la instrucción if...else
152
153    } // fin del método mostrarMensaje
154
155 } // fin de la clase Craps
```

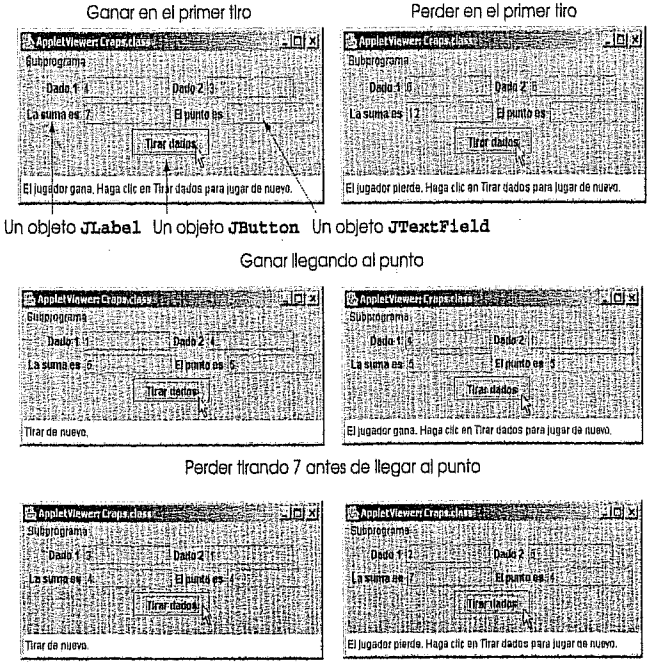


Figura 6.9 Simulación de craps. (Parte 4 de 4.)

http://librosysolucionarios.net

El jugador debe tirar dos dados en todos los tiros. Al ejecutar el applet, haga clic en el botón **Tirar dados** para jugar. La barra de estado en la parte inferior de la ventana del `appletviewer` muestra el resultado de cada tiro. Las capturas de pantalla muestran cuatro ejecuciones separadas del applet (una victoria en el primer tiro, una derrota en el primer tiro, una victoria llegando al punto y una derrota al tirar un 7 antes de llegar al punto.)

Hasta ahora, la mayor parte de las interacciones con el usuario en nuestros programas han sido a través de un cuadro de diálogo de entrada (en donde el usuario puede escribir un valor de entrada para el programa) o cuadro de diálogo de mensaje (en donde se muestra un mensaje al usuario y éste puede hacer clic en **Aceptar** para desaparecer el cuadro). Aunque estos cuadros de diálogo son formas válidas de recibir entrada de un usuario y de mostrar la salida en un programa de Java, sus capacidades son bastante limitadas; un cuadro de diálogo de entrada puede obtener del usuario sólo un valor a la vez, y un cuadro de diálogo de mensaje sólo puede mostrar un mensaje. Sin embargo, por lo general, se requieren más entradas del usuario a la vez (como cuando el usuario introduce su nombre y dirección) o se requiere mostrar diversas piezas de datos (como los valores de los dados, la suma de los dados y el punto, en este ejemplo). Para comenzar nuestra introducción a las interfaces de usuario más elaboradas, mostramos en nuestro programa dos nuevos conceptos de interfaz gráfica de usuario: adjuntar varios componentes de la GUI a un applet y el *manejo de eventos* de la GUI.

Las declaraciones `import` en las líneas 3 a 6 indican los paquetes de los cuales se utilizan clases en este applet. En la línea 3 se especifica que el programa utiliza clases del paquete `java.awt` (específicamente, las clases `Container` y `FlowLayout`). En la línea 4 se especifica que el programa utiliza clases del paquete `java.awt.event`. Este paquete contiene muchos tipos que permiten que un programa procese las interacciones de un usuario con la GUI de un programa. En nuestro programa, utilizamos los tipos `ActionListener` y `ActionEvent` del paquete `java.awt.event`. En la línea 6 se especifica que el programa utiliza clases del paquete `javax.swing` (específicamente, utilizamos las clases `JApplet`, `JButton`, `JLabel` y `JTextField`).

Como lo indicamos anteriormente en el libro, todo programa de Java se basa cuando menos en una declaración de clase que extiende y mejora una declaración de clase existente, por medio de la herencia. Recuerde que las nuevas clases de applets extienden la clase `JApplet`, de manera que puedan heredar sus atributos y comportamientos (campos y métodos) existentes. En la línea 8 se indica que la clase `Craps` hereda de `JApplet` y que *implementa* a la interfaz `ActionListener`. Además de extender una superclase, una clase puede implementar una o más *interfaces*. Una interfaz es un tipo que especifica uno o más métodos (comportamientos), los cuales usted debe declarar en su clase. La interfaz `ActionListener` declara al método `actionPerformed`. Para implementar la interfaz `ActionListener`, tenemos que implementar un método con la primera línea

```
public void actionPerformed( ActionEvent actionEvent )
```

en nuestra clase `Craps` (como se muestra en la línea 66). La tarea de este método es procesar la interacción de un usuario con el objeto `JButton` (llamado **Tirar dados** en la interfaz de usuario) en este applet. Cuando el usuario oprime el botón, el método `actionPerformed` se llama en respuesta a la interacción del usuario. Este proceso de responder a las interacciones de los usuarios se conoce como *manejo de eventos*. El *evento* es la interacción del usuario (por ejemplo, oprimir el botón). El método `actionPerformed` es el *manejador de eventos*, el cual contiene la lógica del juego de Craps en el programa. Hablaremos en breve sobre los detalles relacionados con la interacción del manejo de eventos y el método `actionPerformed`.[8]

El juego de los dados es razonablemente complicado. El jugador puede ganar o perder en el primer tiro, o puede ganar o perder en cualquier tiro subsiguiente. En la línea 11 se crean e inicializan variables que especifican los tres estados de un juego de dados: Juego ganado (`GANADO`), juego perdido (`PERDIDO`) o seguir tirando los dados (`CONTINUAR`). La palabra clave `final` al principio de la declaración indica que éstas son *variables constantes* (variables cuyos valores no pueden cambiarse). Cuando un programa declara una variable `final`, éste debe inicializar la variable antes de usarla, y no puede modificar su valor a partir de entonces. Si la variable

8. En el capítulo 9 hablaremos detalladamente sobre las interfaces. Por ahora, a medida que vaya desarrollando sus propios applets con interfaces gráficas de usuario, imite las características que soportan el manejo de eventos de los componentes de la GUI que presentamos.

es un campo, esta inicialización generalmente ocurre en la declaración de la variable.[9] Los programadores a menudo se refieren a las variables constantes como *constantes*, *constantes con nombre* o *variables de sólo lectura*.

Error común de programación 6.10

Utilizar el valor de una variable `final` sin inicializar o intentar modificar una variable `final` después de inicializarla, son acciones que producen un error de compilación.

Buena práctica de programación 6.4

Por convención, utilice sólo letras mayúsculas (con guiones bajos entre palabras) en los nombres de variables `final`. Este formato hace que las constantes resalten en un programa.

Buena práctica de programación 6.5

Utilizar variables `final` con nombres significativos, en vez de literales (como 2), hace que los programas sean más legibles y fáciles de modificar.

En las líneas 13 a 16 se declaran varios campos que se utilizan en los métodos del applet `Craps`. La variable `primerTiro` es una variable tipo `boolean` que indica si el siguiente tiro de los dados es el primer tiro en el juego actual (`true` cuando el applet empieza a ejecutarse y cuando comienza cada juego nuevo). La variable `sumaDeDados` mantiene la suma de los dados durante el último tiro. La variable `miPunto` almacena el "punto" si el jugador no gana o pierde en el primer tiro. La variable `estadoJuego` lleva el registro del estado actual del juego (`GANADO`, `PERDIDO` o `CONTINUAR`).

En las líneas 19 a 21 se declaran las referencias a los componentes de la GUI que se utilizan en la interfaz gráfica de usuario de este applet. Las referencias `dado1Etiqueta`, `dado2Etiqueta`, `sumaEtiqueta` y `puntoEtiqueta` hacen referencia a un objeto de la clase *`JLabel`*, un componente de la GUI que contiene una cadena de caracteres a mostrar en la pantalla. Generalmente, un objeto `JLabel` indica el propósito de otro componente de la GUI en la pantalla. En las capturas de pantalla de la figura 6.9, los objetos `JLabel` son el texto a la izquierda de cada rectángulo en las primeras dos filas de la interfaz de usuario. Las referencias `dado1Campo`, `dado2Campo`, `sumaCampo` y `puntoCampo` hacen referencia a un objeto de la clase *`JTextField`* (un componente de la GUI que se utiliza para obtener una cadena de información del usuario mediante el teclado, o para mostrar información en la pantalla). Los objetos `JTextField` son los rectángulos a la derecha de cada objeto `JLabel` en las primeras dos filas de la interfaz de usuario. La referencia `tirarBoton` hace referencia a un objeto de la clase *`JButton`*. Cuando el usuario oprime un objeto `JButton`, el programa normalmente responde realizando una tarea (tirando los dados, en este ejemplo). El objeto `JButton` es el rectángulo que contiene las palabras **Tirar dados** en la parte inferior de la interfaz de usuario que se muestra en la figura 6.9. Hemos visto objetos `JButton` en programas anteriores: cada cuadro de diálogo de mensaje y cada cuadro de diálogo de entrada contenían un botón **Aceptar** para desaparecer el cuadro de diálogo o enviar la entrada del usuario al programa. También hemos visto objetos `JTextField` en programas anteriores. Cada cuadro de diálogo de entrada contiene un objeto `JTextField`, en el cual el usuario escribe un valor de entrada.

El método `init` (líneas 24 a 63) crea los objetos de los componentes de la GUI y los adjunta al panel de contenido del applet (como en la figura 6.3). En la línea 27 se inicializa la referencia `contenedor` tipo `Container` con el resultado de la llamada al método `getContentPane` heredado del applet. Recuerde que en la sección 6.4 vimos que el método `getContentPane` devuelve una referencia al panel de contenido del applet, la cual puede usarse para adjuntar componentes de la GUI a la interfaz de usuario del applet.

En la línea 28 se utiliza el método *`setLayout`* de `Container` para especificar el *administrador de esquemas* para la interfaz de usuario del applet. Un administrador de esquemas ordena los componentes de la GUI en un objeto `Container`, para presentarlos en la pantalla. El administrador de esquemas determina la posición y el tamaño de todos los componentes de la GUI adjuntos al contenedor, procesando en consecuencia la mayoría de los detalles del esquema y permitiendo al programador concentrarse en la apariencia visual básica del programa.

9. La inicialización puede ocurrir también en el *constructor* de la clase (que describimos en el capítulo 8), el cual es similar a un método. Proporcionaremos mayores detalles sobre la palabra clave `final` en los capítulos 7 y 8.

`FlowLayout` es el administrador de esquemas más simple. Los componentes de la GUI se colocan de izquierda a derecha, en el orden en el que se adjuntan al objeto `Container` (el panel de contenido del applet, en este ejemplo) con el método `add`. Cuando el administrador de esquemas llega al borde del contenedor, empieza una nueva fila de componentes y continúa con su distribución en esa fila. En la línea 28 se crea un nuevo objeto de la clase `FlowLayout` y lo pasa como el argumento para el método `setLayout`. Generalmente, el esquema se establece antes de que se agreguen componentes de la GUI a un objeto `Container`.[10] Cuando ejecute este applet en el `appletviewer`, cambie el tamaño de la ventana para ver cómo funciona el objeto `FlowLayout` para áreas de applets de distinto tamaño. Elegimos una anchura y altura para este applet que permitiera al objeto `FlowLayout` posicionar los componentes de la GUI según se muestra en la figura 6.9.

Error común de programación 6.11

Si un objeto `Container` no es lo suficientemente grande tal vez no se muestren algunos o ninguno de los componentes de la GUI que tiene adjuntos.

En las líneas 31 a 35, 38 a 42, 45 a 49 y 52 a 56 se crea un objeto `JLabel` y un objeto `JTextField`, y se adjuntan a la interfaz de usuario. Como estos conjuntos de líneas son bastante similares, nos concentraremos en las líneas 31 a 35. En la línea 31 se crea un nuevo objeto `JLabel`, se inicializa con la cadena `"Dado 1"` y se asigna el objeto a la referencia `dado1Etiqueta`. Esta etiqueta precede al objeto `JTextField` correspondiente (llamado `dado1Campo`) en la interfaz de usuario, para que el usuario pueda determinar el propósito del valor que se muestra en el campo de texto. En la línea 32 se adjunta `dado1Etiqueta` al panel de contenido del applet. En la línea 33 se crea un nuevo objeto `JTextField`, se inicializa con `10` caracteres de anchura y se asigna el objeto a la referencia `dado1Campo`. Este objeto `JTextField` muestra el valor del primer dado después de cada tiro de los dados. En la línea 34 se utiliza el método `setEditable` de `JTextField` con el argumento `false`, para indicar que el usuario no puede escribir en el objeto `JTextField` (esto también se conoce como hacer que el objeto `JTextField` *no pueda editarse*). Los campos de texto no editables se muestran con un fondo gris, de manera predeterminada. Un objeto `JTextField` *editable* tiene un fondo blanco (como se ve en los cuadros de diálogo de entrada). En la línea 35 se adjunta `dado1Campo` al panel de contenido del applet. En las líneas 38 a 56 se realizan tareas idénticas para las demás etiquetas y campos de texto.

En la línea 59 se crea un nuevo objeto `JButton`, se inicializa con la cadena `"Tirar dados"` (la cual aparecerá en el botón) y se asigna ese objeto a la referencia `tirarBoton`.

En la línea 60 se especifica que este applet (mediante la palabra clave *`this`*) debe *escuchar* los eventos producidos por el botón `tirarBoton`. La palabra clave `this` permite al applet referirse a sí mismo. (Hablaremos detalladamente sobre `this` en el capítulo 8.) Cuando el usuario interactúa con un componente de la GUI, se envía un evento al applet. Los eventos de la GUI son mensajes que indican que el usuario del programa ha interactuado con uno de los componentes de la GUI del programa. Por ejemplo, cuando usted oprime el objeto `JButton` en la interfaz de usuario de este programa, se envía un mensaje al applet para indicarle que el evento ha ocurrido, notificándole que usted oprimió el botón. Para un objeto `JButton`, el mensaje indica al applet que el usuario *realizó una acción* sobre el objeto `JButton`, lo cual produce una llamada al método `actionPerformed` para procesar la interacción del usuario.

Este estilo de programación se conoce como *programación controlada por eventos*: el usuario interactúa con un componente de la GUI, el programa recibe la notificación del evento y lo procesa. La interacción del usuario con la GUI "controla" el programa. Los métodos que se llaman al ocurrir un evento se conocen como *métodos manejadores de eventos*, o simplemente como *manejadores de eventos*. Cuando ocurre un evento de la GUI en un programa, el mecanismo para manejar eventos crea un objeto que contiene información acerca del evento que acaba de ocurrir, y llama a un método apropiado para manejar eventos (`actionPerformed` para un objeto `JButton`). Antes de que pueda procesarse cualquier evento, cada componente de la GUI debe saber qué objeto en el programa tiene el método manejador de eventos que será llamado cuando ocurra un

10. Cada objeto `Container` puede tener sólo un administrador de esquemas a la vez. Varios objetos `Container` en el mismo programa pueden tener distintos administradores de esquemas. La mayoría de los entornos de programación de java proveen herramientas de diseño de GUIs para ayudar a un programador a que diseñe gráficamente una GUI; después las herramientas generan el código en Java para crear la GUI. En los capítulos 13 y 14 hablaremos sobre varios administradores de esquemas que permiten un control más preciso sobre la distribución de los componentes de la GUI.

http://librosysolucionarios.net

evento. En la línea 60, el método *addActionListener* de `JButton` se utiliza para indicar a `tirarBoton` que el applet (`this`) puede escuchar *eventos de acción*, y que contiene el método `actionPerformed`. A este procedimiento se le llama *registrar el manejador de eventos* con el componente de la GUI. (También nos gusta llamarlo la *línea para empezar a escuchar*, ya que ahora el applet escucha los eventos del botón.) Para responder a un evento de acción, debemos declarar una clase que implemente a `ActionListener` (para ello se requiere que la clase declare también al método `actionPerformed`), y debemos registrar el manejador de eventos con el componente de la GUI. Finalmente, en la línea 61 de `init` se adjunta el objeto `JButton` con el cual `tirarBoton` hace referencia al panel de contenido del applet, completando así la interfaz de usuario.

El método `actionPerformed` (líneas 66 a 115) procesa las interacciones entre el usuario y el objeto `JButton` en la GUI de este applet.[11] En la primera línea del método se indica que `actionPerformed` es un método público que no devuelve nada (`void`) al completar su tarea. El método `actionPerformed` recibe un argumento (un objeto `ActionEvent`) cuando es llamado en respuesta a una acción realizada sobre un componente de la GUI por parte del usuario (en este caso, oprimir el botón `JButton`). El argumento `ActionEvent` contiene información sobre la acción que ocurrió. Este objeto se pasa a `actionPerformed` mediante el mecanismo de manejo de eventos de Java. El método `actionPerformed` es otro método más que Java llama por usted.

Declaramos el método `tirarDados` (líneas 118 a 133) para tirar los dados, calcular y mostrar su suma. El método `tirarDados` no toma argumentos, por lo que tiene una lista de parámetros vacía. Este método devuelve la suma de los dos dados, por lo que se indica el tipo `int` como valor de retorno en el encabezado del método.

El usuario hace clic en el botón **Tirar dados** para realizar su tiro. Esta acción llama al método `actionPerformed` en el applet. En la línea 68 del método `actionPerformed` se hace una llamada a `tirarDados`, el cual elige dos valores aleatorios entre 1 y 6, muestra el valor del primer dado, el valor del segundo dado y la suma de ambos en los primeros tres objetos `JTextField`, respectivamente, y devuelve la suma de los dados. Los valores enteros se convierten en cadenas (líneas 127 a 129) con el método estático *toString* de la clase `Integer`, ya que los objetos `JTextField` sólo pueden mostrar cadenas. Después, el método `actionPerformed` comprueba la variable `primerTiro` tipo `boolean` (línea 71) para determinar si su valor es `true`. De ser así, entonces éste es el primer tiro del juego. La instrucción `switch` anidada de las líneas 73 a 98 determina si el juego se ha ganado o perdido, o si debe continuar con otro tiro. Después del primer tiro, si el juego no ha terminado, el valor de `sumaDeDados` se guarda en `miPunto` (línea 93) y se muestra en el campo de texto `puntoCampo` (línea 94).

En la línea 113 se hace una llamada al método `mostrarMensaje` (declarado en las líneas 136 a 153) para mostrar el estado actual del juego. Este método llama al método *showStatus* del applet para mostrar una cadena en la barra de estado del contenedor del applet. En la línea 140 se muestra

```
Tirar de nuevo.
```

si `estadoJuego` es igual a `CONTINUAR`. En la línea 145 se muestra

```
El jugador gana. Haga clic en Tirar dados para jugar de nuevo.
```

si `estadoJuego` es igual a `GANADO`. En la línea 147 se muestra

```
El jugador pierde. Haga clic en Tirar dados para jugar de nuevo.
```

si `estadoJuego` es igual a `PERDIDO`. Si el juego se acaba, en la línea 149 se asigna el valor `true` a la variable `primerTiro`, indicando que el siguiente tiro del dado será el primer tiro del siguiente juego.

Después el programa espera a que el usuario haga clic de nuevo en el botón **Tirar dados**. Cada vez que el usuario oprime ese botón, el método `actionPerformed` llama al método `tirarDados` para producir un nuevo valor en `sumaDeDados`. Si el tiro actual es la continuación de un juego incompleto, se ejecuta el código de las líneas 102 a 111. En la línea 105, si `sumaDeDados` concuerda con `miPunto`, en la línea 106 se

11. Pronto veremos que hay muchos métodos manejadores de eventos para los diversos componentes de la GUI. Describiremos varios de estos componentes de la GUI y los métodos manejadores de eventos en los capítulos 13 y 14.

asigna el valor `GANADO` a la variable `estadoJuego` y se completa el juego. En la línea 108, si `sumaDeDados` es igual a 7, en la línea 109 se asigna el valor `PERDIDO` a la variable `estadoJuego` y se completa el juego. Al completarse el juego, `mostrarMensaje` muestra un mensaje apropiado y el usuario puede hacer clic en el botón **Tirar dados** para iniciar un juego nuevo. A lo largo del programa, los campos de texto `dado1Campo`, `dado2Campo` y `sumaCampo` se actualizan con los nuevos valores de los dados y la suma en cada tiro, y el valor de `puntoCampo` se actualiza cada vez que empieza un nuevo juego.

Observe el uso de los diversos mecanismos de control de programas que hemos descrito. El programa de los dados utiliza cuatro métodos (`init`, `actionPerformed`, `tirarDados` y `mostrarMensaje`) y las instrucciones `switch`, `if...else` e `if` anidadas. Observe también el uso de varias etiquetas `case` en la instrucción `switch` para ejecutar las mismas instrucciones para las sumas de 7 y 11 (líneas 76 a 80), y para las sumas de 2, 3 y 12 (líneas 83 a 88). El mecanismo manejador de eventos también actúa como una forma de control del programa. En este programa, el manejo de eventos permite la repetición controlada por el usuario: cada vez que el usuario hace clic en **Tirar dados**, el programa tira los dados de nuevo. En el ejercicio 6.46 investigaremos varias características interesantes del juego de los dados.

6.9 Alcance de las declaraciones

En los ejemplos anteriores hemos visto declaraciones de varias entidades de Java, como clases, métodos, variables y parámetros. Las declaraciones introducen nombres que pueden usarse para referirse a dichas entidades. El *alcance* de una declaración es la porción del programa que puede referirse a una entidad por su nombre. Dicha declaración se encuentra *dentro del alcance* para esa porción del programa. En esta sección ofrecemos una introducción a varias cuestiones importantes, relacionadas con el alcance.[12]

Las reglas básicas de alcance son las siguientes:

1. El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece la declaración. (En el capítulo 8 hablaremos sobre los constructores, que son similares a los métodos. El alcance de la declaración de un parámetro de constructor es el cuerpo de ese constructor.)
2. El alcance de la declaración de una variable local es a partir del punto en el que la declaración aparece en el bloque, hasta el término de ese bloque.
3. El alcance de una etiqueta en una instrucción `break` o `continue` etiquetada (que presentamos en la sección 5.8) es la instrucción encerrada por la instrucción etiquetada (es decir, el cuerpo de la instrucción etiquetada).
4. El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for` y las demás expresiones en el encabezado.
5. El alcance de un método o campo de una clase es el cuerpo completo de la clase. Esto permite a los métodos de una clase utilizar nombres simples para llamar a otros métodos declarados en la clase, o heredados por esa clase (como los métodos que nuestras clases heredan de la clase `JApplet`), y para acceder a los campos declarados en la clase.[13]

Cualquier bloque puede contener declaraciones de variables o referencias. Cuando los bloques se anidan en el cuerpo de un método y un identificador declarado en un bloque externo tiene el mismo nombre que un identificador declarado en un bloque interno, el compilador genera un error de sintaxis, indicando que la variable ya está declarada. Si una variable local o parámetro en un método tienen el mismo nombre que un campo, el campo se "oculta" hasta que el bloque termina de ejecutarse. Java llama a esto *sombreado*. En el capítulo 8 veremos cómo tener acceso a los campos sombreados.

12. Vea el documento *Java Language Specification, Sección 6.3: Scope of a Declaration* (`java.sun.com/docs/books/jls/second_edition/html/names.doc.html#103228`), para obtener los detalles completos sobre los alcances de Java.

13. En el capítulo 8 veremos que los métodos `static` son una excepción a esta regla. Ésta es la razón por la cual utilizamos applets para crear programas que contengan varias declaraciones de métodos. Una vez que presentemos los detalles completos sobre las clases en los capítulos 8 a 10, utilizaremos principalmente aplicaciones para implementar los ejemplos en el resto del libro.

http://librosysolucionarios.net

Error común de programación 6.12

Declarar una entidad en un bloque interno con el mismo nombre que una entidad en un bloque externo del mismo método, produce un error de sintaxis.

Buena práctica de programación 6.6

Evite utilizar los mismos nombres para campos y variables locales. El uso de nombres diferentes ayuda a los lectores de su programa a distinguir las variables usadas en distintas partes de la declaración de una clase.

El applet de la figura 6.10 demuestra las cuestiones relacionadas con el alcance de los campos y las variables locales. Éste es nuestro primer ejemplo de un applet que implementa el método **start** (líneas 24 a 37). Recuerde que en la sección 3.3 vimos que cuando un contenedor de applets carga un applet, primero crea una instancia de la clase del applet. Después llama a los métodos **init**, **start** y **paint** del applet (**paint** no se utiliza en este ejemplo). El método **start** siempre se declara con el encabezado que se muestra en la línea 24.

```
1  // Fig. 6.10: Alcance.java
2  // Un ejemplo del alcance.
3  import java.awt.Container;
4
5  import javax.swing.*;
6
7  public class Alcance extends JApplet {
8     JTextArea areaSalida;
9
10    // campo que es accesible para todos los métodos de esta clase
11    int x = 1;
12
13    // crear GUI del applet
14    public void init()
15    {
16       areaSalida = new JTextArea();
17       Container contenedor = getContentPane();
18       contenedor.add( areaSalida );
19
20    } // fin del método init
21
22    // el método start se llama después de que init termina; start llama a
23    // los métodos usarLocal and usarCampo
24    public void start()
25    {
26       int x = 5;    // variable local en método start que sombrea el campo x
27
28       areaSalida.append( "x local en start es " + x );
29
30       usarLocal();  // usarLocal tiene x local
31       usarCampo();  // useInstance modifica el campo x de Alcance
32       usarLocal();  // usarLocal inicializa la variable local x
33       usarCampo();  // Campo x de Alcance retiene su valor
34
35       areaSalida.append( "\n\nx local en start es " + x );
36
37    } // fin del método start
38
39    // usarLocal crea e inicializa la variable local x durante cada llamada
40    public void usarLocal()
41    {
```

Figura 6.10 Ejemplo del alcance. (Parte 1 de 2.)

http://librosysolucionarios.net


```
42       int x = 25;  // se inicializa cada vez que se hace una llamada a
         usarLocal
43
44       areaSalida.append( "\n\nx local en usarLocal es " + x +
45          " después de entrar a usarLocal" );
46       ++x;
47       areaSalida.append( "\nx local en usarLocal es " + x +
48          " después de salir de usarLocal" );
49
50    } // fin del método usarLocal
51
52    // usarCampo modifica el campo x de Alcance durante cada llamada
53    public void usarCampo()
54    {
55       areaSalida.append( "\n\nel campo x es " + x +
56          " al entrar a usarCampo" );
57       x *= 10;
58       areaSalida.append( "\nel campo x es " + x +
59          " al salir de usarCampo" );
60
61    } // fin el método usarCampo
62
63 } // fin de la clase Alcance
```

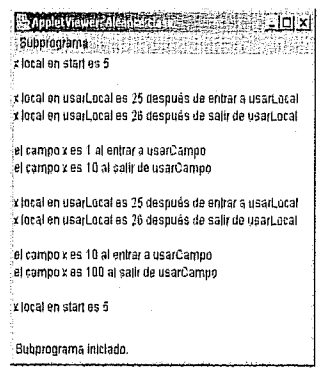


Figura 6.10 Ejemplo del alcance. (Parte 2 de 2.)

En la línea 11 se declara e inicializa el campo **x** con 1. Este campo se sombrea (oculta) en cualquier bloque (o método) que declare una variable llamada **x**. El método **start** declara una variable local **x** (línea 26) y la inicializa con 5. El valor de esta variable se anexa al área de texto **areaSalida** para mostrar que el campo **x** está sombreado en **start**. El programa declara otros dos métodos: **usarLocal** (líneas 40 a 50) y **usarCampo** (líneas 53 a 61), cada uno de los cuales no tiene argumentos y no devuelve resultados. El método **start** llama a cada método dos veces (líneas 30 a 33). El método **usarLocal** declara a la variable local **x** (línea 42). Cuando **usarLocal** se llama por primera vez (línea 30), crea la variable local **x** y la inicializa con 25 (línea 42), anexa el valor de **x** a **areaSalida** (líneas 44 y 45), incrementa **x** (línea 46) y anexa el valor de **x** a **areaSalida** otra vez (línea 47). Cuando **usarLocal** se llama una segunda vez (línea 32), vuelve a crear la variable local **x** y la inicializa con 25, de forma que la salida de cada llamada a **usarLocal** es idéntica.

El método **usarCampo** no declara ninguna variable. Por lo tanto, cuando hace referencia a la variable **x**, se utiliza el campo **x** (línea 11). Cuando el método **usarCampo** se llama por primera vez (línea 31), anexa el valor del campo **x** a **areaSalida** (líneas 55 y 56), multiplica el campo **x** por 10 (línea 57) y anexa el valor del campo **x** a **areaSalida** otra vez (línea 58) antes de regresar. La siguiente vez que se llama al método

`usarCampo` (línea 33), el campo tiene su valor modificado, `10`. Finalmente, en el método `start` el programa anexa el valor de la variable local `x` a `areaSalida` otra vez (línea 35), para mostrar que ninguna de las llamadas al método modificó a la variable local `x` de `start`, ya que todos los métodos hicieron referencia a las variables en otros alcances.

6.10 Métodos de la clase `JApplet`

Hemos escrito muchos applets hasta este punto, pero no hemos hablado todavía sobre los métodos clave de la clase `JApplet` que son llamados por el contenedor de applets durante la ejecución de un applet. La figura 6.11 muestra los métodos clave de la clase `JApplet`, especifica cuándo se llaman y explica el propósito de cada uno de ellos. Con la excepción del método `paint`, estos métodos de `JApplet` son declarados por la API de Java de tal forma que no hacen nada, a menos que usted proporcione una declaración en la clase de su applet. El método `paint` original se asegura de que los gráficos y los componentes de la GUI se muestren correctamente. Si usted desea redefinir uno de estos métodos en un applet, *debe* utilizar el encabezado del método apropiado, como se muestra en la figura 6.11. De lo contrario, el contenedor de applets no llamará a su versión del método durante la ejecución del applet. El proceso de declarar los métodos como se describe aquí se conoce como *sobrescribir* el método original; la nueva declaración del método reemplaza a la declaración heredada. El contenedor de applets llamará a la versión sobrescrita de un método para su applet antes de intentar llamar a las versiones predeterminadas que hereda de `JApplet`. El proceso de sobrescritura de métodos se describe con detalle en el capítulo 9.



Error común de programación 6.13

Si proporciona una declaración para uno de los métodos `init`, `start`, `paint`, `stop` o `destroy` de `JApplet`, que no concuerde con el encabezado del método correspondiente que se muestra en la figura 6.11, se generará un método que no será llamado automáticamente durante la ejecución del applet.

El método `repaint` también es de interés para muchos programadores de applets. El método `paint` del applet normalmente es llamado por el contenedor de applets. ¿Qué pasaría si usted quisiera cambiar la apariencia del applet, en respuesta a las interacciones del usuario con el applet? En dichas situaciones, tal vez sea

Método	Cuándo es llamado este método y su propósito
`public void init()`	Este método es llamado una vez por el contenedor de applets cuando se carga un applet para ejecutarse. Lleva a cabo la inicialización de un applet. Las acciones típicas que se realizan aquí son inicializar campos, crear componentes de la GUI, cargar los sonidos a reproducir, cargar las imágenes a mostrar (vea el capítulo 19) y crear subprocesos (vea el capítulo 16).
`public void start()`	Este método se llama después de que el método `init` termina de ejecutarse. Además, si el navegador visita otro sitio Web y después regresa a la página de HTML en la que reside el applet, el método `start` se llama de nuevo. Este método lleva a cabo todas las tareas que deben completarse cuando el applet se carga por primera vez, y que deben llevarse a cabo cada vez que se vuelva a visitar la página de HTML en la que reside el applet. Las acciones típicas que se realizan aquí incluyen el empezar una animación (vea el capítulo 19) y empezar otros subprocesos de ejecución (vea el capítulo 16).
`public void paint( Graphics g )`	Este método de dibujo se llama después de que el método `init` termina de ejecutarse y el método `start` ha comenzado. También se llama cada vez que el applet necesita volver a pintarse. Por ejemplo, si el usuario cubre el applet con otra ventana abierta en la pantalla y después descubre el applet, se hace una llamada al método `paint`. Las acciones típicas que se realizan aquí incluyen el dibujar con el objeto `g` tipo `Graphics` que se pasa al método `paint` mediante el contenedor de applets.

Figura 6.11 Métodos `JApplet` que son llamados por el contenedor de applets durante la ejecución de un applet. (Parte 1 de 2.)

Método	Cuándo es llamado este método y su propósito
`public void stop()`	Este método se llama cuando el applet debe detener su ejecución; normalmente, cuando el usuario del navegador sale de la página HTML en la que reside el applet. El método realiza todas las tareas requeridas para suspender la ejecución del applet. Las acciones típicas que se realizan aquí son detener la ejecución de las animaciones y los subprocesos.
`public void destroy()`	Este método se llama cuando el applet se va a quitar de memoria; normalmente, cuando el usuario del navegador sale de la sesión de navegación (es decir, cuando cierra todas las ventanas del navegador). El método realiza todas las tareas requeridas para destruir los recursos asignados al applet.

Figura 6.11 Métodos `JApplet` que son llamados por el contenedor de applets durante la ejecución de un applet. (Parte 2 de 2.)

conveniente que llame al método `paint` directamente. Sin embargo, para llamar a `paint` debemos pasarle el parámetro `Graphics` que espera. No tenemos un objeto `Graphics` a nuestra disposición para pasarlo a `paint` (ni sabemos cómo crear uno). Por esta razón, la clase `JApplet` proporciona el método `repaint`. La instrucción

```
repaint();
```

obtiene el objeto `Graphics` por usted y llama a otro método, llamado `update`, que a su vez llama al método `paint` y le pasa el objeto `Graphics`. Describiremos detalladamente el método `repaint` en el capítulo 19.

6.11 Sobrecarga de métodos

Java permite que varios métodos con el mismo nombre se declaren en la misma clase, siempre y cuando estos métodos tengan distintos conjuntos de parámetros (los cuales se determinan mediante el número y los tipos de los parámetros). Esta técnica se conoce como *sobrecarga de métodos*. Cuando se llama a un método sobrecargado, el compilador de Java selecciona el método apropiado examinando el número y los tipos de los argumentos en la llamada. La sobrecarga de métodos se utiliza comúnmente para crear varios métodos con el mismo nombre que realizan tareas similares, pero en tipos de datos distintos. La figura 6.12 utiliza el método sobrecargado `cuadrado` para calcular el cuadrado de un `int` y el cuadrado de un `double`.



Buena práctica de programación 6.7

Sobrecargar métodos que realizan tareas estrechamente relacionadas, pero no idénticas, puede hacer que los programas sean más legibles y comprensibles.

Los métodos sobrecargados se distinguen por su *firma*: una combinación del nombre del método, el número y tipo de sus parámetros. Si el compilador de Java analizara solamente los nombres de los métodos durante la compilación, el código de la figura 6.12 sería ambiguo; el compilador no sabría cómo distinguir entre los dos métodos `cuadrado` (líneas 22 a 29 y 32 a 39). El compilador utiliza nombres más largos "deformados" o "decorados", que incluyen el nombre original del método, los tipos de cada parámetro y el orden exacto de los parámetros, para determinar si los métodos en una clase son únicos en esa clase.

Por ejemplo, en la figura 6.12 el compilador podría utilizar el nombre lógico "cuadrado de int" para el método `cuadrado` que especifica un parámetro `int`, y "cuadrado de double" para el método `cuadrado` que especifica un parámetro `double`. Si la declaración de `metodo1` empieza como

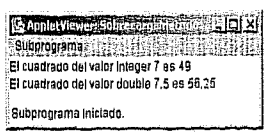
```
void metodo1( int a, float b )
```

entonces el compilador podría utilizar el nombre lógico "metodo1 de int y float". Si los parámetros se especificaran como

```
void metodo1( float a, int b )
```

http://librosysolucionarios.net

```
1  // Fig. 6.12: SobrecargaMetodos.java
2  // Uso de métodos sobrecargados
3  import java.awt.Container;
4
5  import javax.swing.*;
6
7  public class SobrecargaMetodos extends JApplet {
8
9     // crear GUI y llamar a cada uno de los métodos cuadrado
10    public void init()
11    {
12       JTextArea areaSalida = new JTextArea();
13       Container contenedor = getContentPane();
14       contenedor.add( areaSalida );
15
16       areaSalida.setText( "El cuadrado del valor integer 7 es " + cuadrado( 7 ) +
17          "\nEl cuadrado del valor double 7.5 es " + cuadrado( 7.5 ) );
18
19    } // fin del método init
20
21    // método cuadrado con argumento int
22    public int cuadrado( int intValor )
23    {
24       System.out.println( "Se llamó a cuadrado con argumento int: " +
25          intValor );
26
27       return intValor * intValor;
28
29    } // fin del método cuadrado con argumento int
30
31    // método cuadrado con argumento double
32    public double cuadrado( double doubleValor )
33    {
34       System.out.println( "Se llamó a cuadrado con argumento double: " +
35          doubleValor );
36
37       return doubleValor * doubleValor;
38
39    } // fin del método cuadrado con argumento double
40
41 } // fin de la clase SobrecargaMetodos
```




```
Se llamó a cuadrado con argumento int: 7
Se llamó a cuadrado con argumento double: 7.5
```

Figura 6.12 Declaraciones de métodos sobrecargados.


```
1  // Fig. 6.13: SobrecargaMetodos.java
2  // Métodos sobrecargados con firmas idénticas.
3  import javax.swing.JApplet;
4
5  public class SobrecargaMetodos extends JApplet {
6
7     //  declaración del método cuadrado con argumento int
8     public int cuadrado( int x )
9     {
10       return x * x;
11    }
12
13    // la segunda declaración del método cuadrado
14    // con argumento int produce un error de sintaxis
15    public double cuadrado( int y )
16    {
17       return y * y;
18    }
19
20 } // fin de la clase SobrecargaMetodos
```

```
SobrecargaMetodos.java:15: cuadrado( int ) is already defined in
SobrecargaMetodos public double cuadrado( int y )
       ^
1 error
```

Figura 6.13 Las declaraciones de métodos sobrecargados con listas de parámetros idénticas ocasionan errores de compilación.

entonces el compilador podría utilizar el nombre lógico "metodo1 de float e int". Observe que el orden de los tipos de los parámetros es importante para el compilador. El compilador considera que los dos encabezados anteriores de `metodo1` son distintos.

Observe que, al hablar sobre los nombres lógicos de los métodos utilizados por el compilador, no mencionamos los tipos de valores de retorno de los métodos, ya que éstos no pueden distinguirse en base al tipo de valor de retorno. El programa de la figura 6.13 ilustra los errores de compilador generados cuando dos métodos tienen la misma firma y distintos tipos de valor de retorno. Los métodos sobrecargados pueden tener distintos valores de retorno, pero deben tener distintas listas de parámetros. Además, los métodos sobrecargados no necesitan tener el mismo número de parámetros.



Error común de programación 6.14

Declarar métodos sobrecargados con listas idénticas de parámetros es un error de sintaxis.

6.12 Recursividad

Los programas que hemos visto hasta ahora se estructuran generalmente como métodos que se llaman entre sí, en forma disciplinada y jerárquica. Sin embargo, para algunos problemas es necesario hacer que un método se llame a sí mismo. Un *método recursivo* es un método que se llama a sí mismo, ya sea directa o indirectamente, a través de otro método. La recursividad es un tema importante que se describe detalladamente en cursos de ciencia computacional avanzados. En esta sección y en la que sigue, presentaremos ejemplos simples de recursividad. En este libro incluimos un análisis extenso sobre la recursividad. En la figura 6.18 (al final de la sección 6.14) se sintetizan los ejemplos y ejercicios de recursividad que se incluyen en el libro.

Primero consideraremos a la recursividad en forma conceptual. Luego examinaremos diversos programas que contienen métodos recursivos. Las metodologías recursivas para solucionar problemas tienen una variedad

http://librosysolucionarios.net

de elementos en común. Cuando se llama a un método recursivo para solucionar un problema, el método es en realidad capaz de resolver sólo el(los) caso(s) más simple(s), o *caso(s) base*. Si se llama al método con un caso base, el método devuelve un resultado. Si el método se llama con un problema más complejo, el método divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver (el caso base) y una pieza que el método no sabe cómo resolver. Para que la recursividad sea factible, la segunda pieza debe asemejarse al problema original, pero debe ser una versión ligeramente más simple o pequeña de él. Como este nuevo problema se ve como el problema original, el método llama a una copia nueva de sí mismo para trabajar en el problema más pequeño; a este procedimiento se le conoce como una *llamada recursiva* y también como el *paso de recursividad*. Este paso de recursividad normalmente incluye una instrucción `return`, ya que su resultado se combinará con la porción del problema que el método sabía cómo resolver, para formar un resultado que se pasará de vuelta al método que originalmente hizo la llamada.

El paso de recursividad se ejecuta mientras la llamada original al método aún sigue activa (es decir, mientras no haya terminado de ejecutarse). Además, puede producir como resultado muchas llamadas recursivas, a medida que el método divide cada nuevo subproblema en dos piezas conceptuales. Para que la recursividad termine en cierto momento, cada vez que el método se llama a sí mismo con una versión ligeramente más simple del problema original, la secuencia de problemas más y más pequeños debe convergir en el caso base. En ese punto, el método reconoce el caso base y devuelve un resultado a la copia anterior del método. Se lleva a cabo una secuencia de regresos (mediante instrucciones `return`) hasta que la llamada al método original devuelve el resultado final al método que hizo la llamada. Este proceso parece ser complejo, en comparación con el método convencional para solucionar problemas que hemos llevado a cabo hasta este punto.

Cálculos recursivos del factorial

Como un ejemplo del funcionamiento de los conceptos de recursividad, escribiremos un programa recursivo para realizar un popular cálculo matemático. Considere el factorial de un entero n no negativo, que se escribe como $n!$ y que es el producto de

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

1! es igual a 1 y 0! se define como 1. Por ejemplo, 5! es el producto de $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

El factorial de un entero, `numero`, mayor o igual a 0, puede calcularse *iterativamente* (no recursivamente) utilizando la instrucción `for` de la siguiente manera:

```
factorial = 1;

for ( int contador = numero; contador >= 1; contador-- )
   factorial *= contador;
```

Para llegar a una declaración recursiva del método del factorial, podemos observar la siguiente relación:

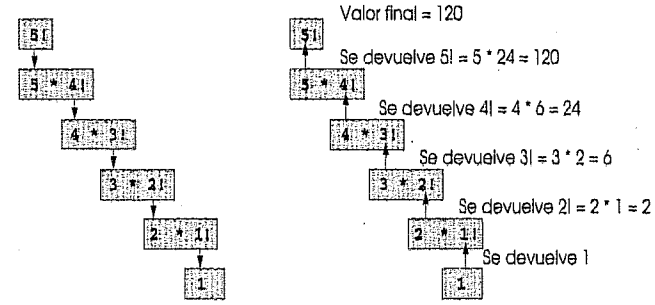
$$n! = n \cdot (n-1)!$$

Por ejemplo, 5! es evidentemente igual a $5 \cdot 4!$, como se muestra mediante las siguientes ecuaciones:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La evaluación de 5! procedería como se muestra en la figura 6.14. En la figura 6.14(a) se muestra cómo la sucesión de llamadas recursivas procede hasta que 1! se evalúa como 1, con lo cual termina la recursividad. En la figura 6.14(b) se muestran los valores devueltos de cada llamada recursiva al método que hizo la llamada, hasta que el valor final se calcula y se devuelve.

La figura 6.15 utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 10. El método recursivo `factorial` (líneas 26 a 36) evalúa primero para determinar si una condición de terminación (línea 29) es `true`. Si `numero` es menor o igual que 1 (el caso base), `factorial` devuelve 1, no se necesita más recursividad y el método regresa. Si `numero` es mayor que 1, en la línea 34 se expresa el problema como el producto de `numero` y una llamada recursiva a `factorial`, evaluando el factorial de `numero - 1`, el cual es un problema ligeramente más simple que el cálculo original, `factorial ( numero )`.



(a) Secuencia de llamadas recursivas (b) Valores devueltos de cada llamada recursiva

Figura 6.14 Evaluación recursiva de 5!

```
1  // Fig. 6.15: PruebaFactorial.java
2  // Método recursivo para el factorial.
3  import java.awt.*;
4
5  import javax.swing.*;
6
7  public class PruebaFactorial extends JApplet {
8     JTextArea areaSalida;
9
10    // crear GUI y calcular factoriales del 0-10
11    public void init()
12    {
13       areaSalida = new JTextArea();
14
15       Container contenedor = getContentPane();
16       contenedor.add( areaSalida );
17
18       // calcular los factoriales de 0 hasta 10
19       for ( long contador = 0; contador <= 10; contador++ )
20          areaSalida.append( contador + "! = " +
21             factorial( contador ) + "\n" );
22
23    } // fin del método init
24
25    // declaración recursiva del método factorial
26    public long factorial( long numero )
27    {
28       // caso base
29       if ( numero <= 1 )
30          return 1;
31
32       // paso recursivo
33       else
34          return numero * factorial( numero - 1 );
35
```

Figura 6.15 Cálculos del factorial con un método recursivo. (Parte 1 de 2.)

http://librosysolucionarios.net

```
36       } // fin del método factorial
37
38    } // fin de la clase PruebaFactorial
```

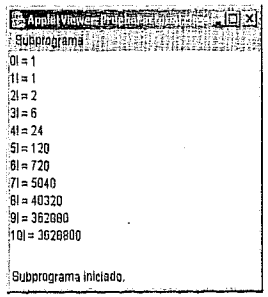


Figura 6.15 Cálculos del factorial con un método recursivo. (Parte 2 de 2.)

El método `factorial` (líneas 26 a 36) recibe un parámetro de tipo `long` y devuelve un resultado del mismo tipo. Como puede verse en la figura 6.15, los valores de los factoriales se hacen grandes rápidamente. Seleccionamos el tipo `long` (que puede representar enteros relativamente grandes) para que el programa pudiera calcular factoriales mayores de 20! Desafortunadamente, el método `factorial` produce valores grandes tan rápidamente que los valores de los factoriales pronto se exceden del valor máximo que puede almacenarse en una variable `long`. Debido a las restricciones en los tipos integrales, tal vez sería necesario utilizar variables de tipo `float` y `double` para calcular factoriales de números más grandes. Esta situación apunta a una debilidad en la mayoría de los lenguajes de programación: a saber, que los lenguajes no se extienden fácilmente para enfrentar los requerimientos únicos de varias aplicaciones. Como veremos en el capítulo 9, Java es un lenguaje extensible que nos permite crear enteros arbitrariamente grandes si lo deseamos. De hecho, el paquete `java.math` proporciona las clases `BigInteger` y `BigDecimal` explícitamente para cálculos matemáticos de precisión arbitraria que no pueden representarse con los tipos primitivos de Java.



Error común de programación 6.15

Si se omite el caso base o se escribe el paso de recursividad incorrectamente, de manera que no converja en el caso base, se producirá una recursividad infinita, lo que eventualmente agotará la memoria. Este error es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva).

6.13 Ejemplo del uso de la recursividad: La serie de Fibonacci

La serie de Fibonacci,

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números anteriores de Fibonacci. Esta serie ocurre en la naturaleza y, en especial, describe una forma de espiral. La proporción de números Fibonacci sucesivos converge en un valor constante de 1.618..., un número que ha sido denominado la proporción dorada o la media dorada. Los humanos tienden a encontrar que la media dorada es estéticamente placentera. Los arquitectos a menudo diseñan ventanas, cuartos y edificios cuya longitud y anchura se encuentran dentro de la proporción de la media dorada. Las tarjetas postales se diseñan a menudo con una proporción de anchura y altura dentro de la media dorada.

La serie de Fibonacci puede definirse recursivamente, de la siguiente manera:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n−1)+fibonacci(n−2)

http://librosysolucionarios.net

Observe que hay dos casos base para el cálculo de Fibonacci: fibonacci(0) se define como 0, y fibonacci(1) se define como 1. El applet de la figura 6.16 calcula el $i^{ésimo}$ número Fibonacci en forma recursiva, utilizando el método `fibonacci` (líneas 62 a 72). El applet permite al usuario introducir un entero en un campo de texto. El valor introducido indica el $i^{ésimo}$ número Fibonacci a calcular. Cuando el usuario oprime *Intro*, el método `actionPerformed` (líneas 43 a 59) se ejecuta en respuesta al evento de la interfaz de usuario, y llama al método `fibonacci` para calcular el número Fibonacci especificado. Estos números tienden a hacerse grandes rápidamente; por lo tanto, utilizamos `long` como el tipo del parámetro y el tipo de valor de retorno de `fibonacci`. En la figura 6.16, las capturas de pantalla muestran los resultados de calcular varios números Fibonacci.

Una vez más, el método `init` (líneas 13 a 40) crea los componentes de la GUI y los adjunta al panel de contenido del applet. El administrador de esquemas para el panel de contenido se establece en `FlowLayout` en la línea 17.

```
1  // Fig. 6.16: PruebaFibonacci.java
2  // Método recursivo fibonacci.
3  import java.awt.*;
4  import java.awt.event.*;
5
6  import javax.swing.*;
7
8  public class PruebaFibonacci extends JApplet implements ActionListener {
9     JLabel numeroEtiqueta, resultadoEtiqueta;
10    JTextField numeroCampo, resultadoCampo;
11
12    // configurar la GUI del applet
13    public void init()
14    {
15       // obtener el panel de contenido y establecer su esquema como FlowLayout
16       Container contenedor = getContentPane();
17       contenedor.setLayout( new FlowLayout() );
18
19       // crear numeroEtiqueta y adjuntarlo al panel de contenido
20       numeroEtiqueta = new JLabel( "Escriba un entero y oprima Intro" );
21       contenedor.add( numeroEtiqueta );
22
23       // crear numeroCampo y adjuntarlo al panel de contenido
24       numeroCampo = new JTextField( 10 );
25       contenedor.add( numeroCampo );
26
27       // registrar este applet como ActionListener de numeroCampo
28       numeroCampo.addActionListener( this );
29
30       // crear resultadoEtiqueta y adjuntarlo al panel de contenido
31       resultadoEtiqueta = new JLabel( "El valor de Fibonacci es" );
32       contenedor.add( resultadoEtiqueta );
33
34       // crear numeroCampo, hacerlo no editable
35       // y adjuntarlo al panel de contenido
36       resultadoCampo = new JTextField( 15 );
37       resultadoCampo.setEditable( false );
38       contenedor.add( resultadoCampo );
39
40    } // fin del método init
41
```

Figura 6.16 Números Fibonacci generados con un método recursivo. (Parte 1 de 3.)

```
42    // obtener entrada del usuario y llamar al método fibonacci
43    public void actionPerformed( ActionEvent event )
44    {
45       long numero, valorFibonacci;
46
47       // obtener entrada del usuario y convertirla en long
48       numero = Long.parseLong( numeroCampo.getText() );
49
50       showStatus( "Calculando ..." );
51
52       // calcular valor de fibonacci para el número introducido por el usuario
53       valorFibonacci = fibonacci( numero );
54
55       // indicar que el procesamiento se completó y mostrar el resultado
56       showStatus( "Listo." );
57       resultadoCampo.setText( Long.toString( valorFibonacci ) );
58
59    } // fin del método actionPerformed
60
61    // declaración recursiva del método fibonacci
62    public long fibonacci( long n )
63    {
64       // caso base
65       if ( n == 0 || n == 1 )
66          return n;
67
68       // paso recursivo
69       else
70          return fibonacci( n - 1 ) + fibonacci( n - 2 );
71
72    } // fin del método fibonacci
73
74 } // fin de la clase PruebaFibonacci
```

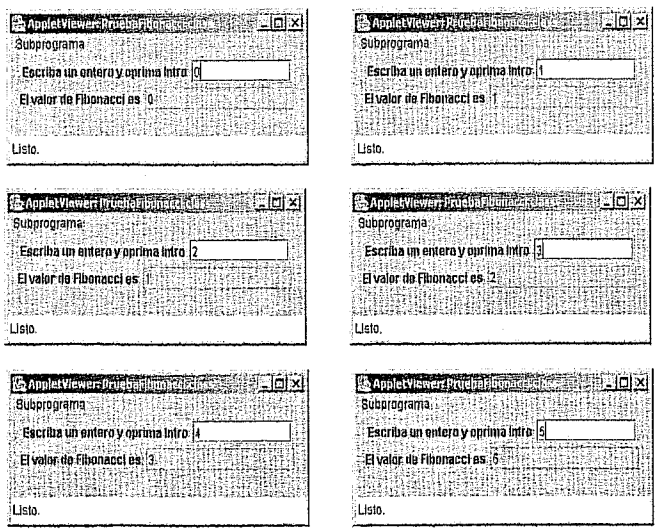


Figura 6.16 Números Fibonacci generados con un método recursivo. (Parte 2 de 3.)

http://librosysolucionarios.net

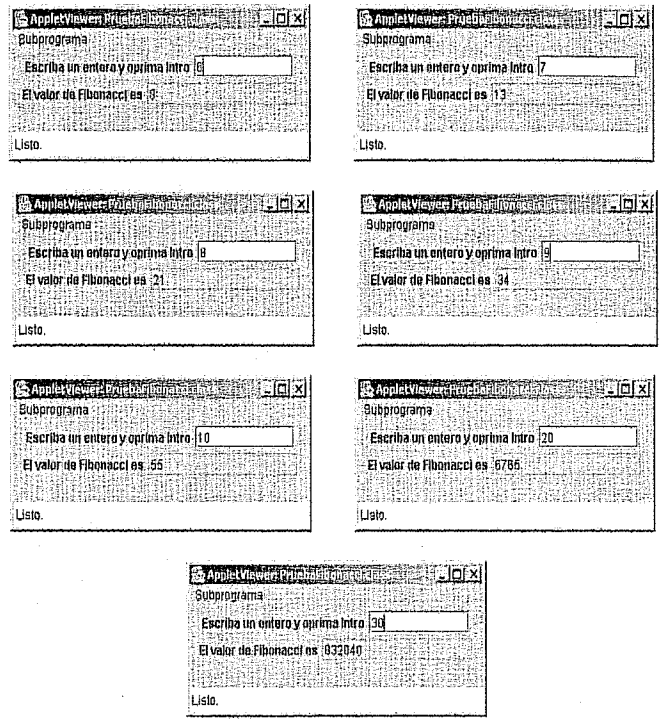


Figura 6.16 Números Fibonacci generados con un método recursivo. (Parte 3 de 3.)

El manejo de eventos en este ejemplo es similar al manejo de eventos del applet `Craps` de la figura 6.9. En la línea 28 se especifica que este applet (mediante `this`) debe escuchar los eventos del campo de texto `numeroCampo`. Recuerde que en la sección 6.8 vimos que la palabra clave `this` permite al applet referirse a sí mismo. Por lo tanto, en la línea 28 el applet indica a `numeroCampo` que el applet debe recibir notificación (mediante una llamada a su método `actionPerformed`) cuando ocurra un evento de acción en el objeto `numeroCampo`. En este ejemplo, el usuario oprime la tecla *Intro* (*Enter*) mientras escribe en el objeto `numeroCampo` para generar el evento de acción. Después se envía un mensaje al applet (es decir, se llama a un método en el applet, en este caso `actionPerformed`) indicando que el usuario del programa ha interactuado con uno de los componentes de la GUI del programa (`numeroCampo`). La instrucción para registrar el applet de manera que escuche los eventos de `numeroCampo` se compilará solamente si la clase del applet implementa también a `ActionListener` (línea 8).

La llamada al método `fibonacci` (línea 53) desde `actionPerformed` no es una llamada recursiva, pero todas las llamadas subsecuentes a `fibonacci` que se realizan desde el cuerpo de `fibonacci` (línea 70) son recursivas. Cada vez que se hace una llamada a `fibonacci`, éste evalúa inmediatamente los casos base: `n` igual a `0` o `1`. Si esta condición es verdadera, `fibonacci` devuelve `n`: `fibonacci(0)` es `0` y `fibonacci(1)` es `1`. Curiosamente, si `n` es mayor que `1` el paso de recursividad genera dos llamadas recursivas, cada una para un problema ligeramente más simple que la llamada original a `fibonacci`. En la figura 6.17 se muestra cómo el método `fibonacci` evalúa `fibonacci(3)`.

En la figura 6.17 se producen ciertas cuestiones interesantes acerca del orden en el que los compiladores de Java evalúan los operandos de los operadores. Este orden es ligeramente distinto al orden en el que se aplican los operadores a sus operandos; a saber, el orden dictado por las reglas de precedencia de los operadores. Con base en la figura 6.17, parece ser que mientras se está evaluando `fibonacci(3)` se harán dos llamadas

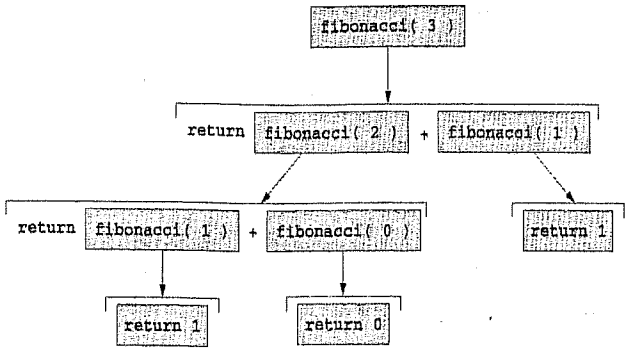


Figura 6.17 Conjunto de llamadas recursivas para `fibonacci(3)`.

recursivas: `fibonacci(2)` y `fibonacci(1)`. Pero ¿en qué orden se harán estas llamadas? La mayoría de los programadores suponen que los operandos se evaluarán de izquierda a derecha. En Java, esta suposición es correcta.[14] El lenguaje Java especifica que el orden de evaluación de los operandos es de izquierda a derecha. Por lo tanto, las llamadas al método son, de hecho, `fibonacci(2)` primero y `fibonacci(1)` después.

Buena práctica de programación 6.8

No escriba expresiones que dependan del orden de evaluación de los operandos de un operador. El uso de dichas expresiones a menudo produce como resultado programas que son difíciles de leer, depurar, modificar y mantener.

Hay que tener cuidado con los programas recursivos como el que utilizamos aquí para generar números Fibonacci. Cada invocación al método `fibonacci` que no concuerda con uno de los casos base (0 o 1) produce dos llamadas recursivas más al método `fibonacci`. Por consecuencia, este conjunto de llamadas recursivas puede salirse de control rápidamente. Para calcular el valor 20 de Fibonacci con el programa de la figura 6.16 se requieren 21,891 llamadas al método `fibonacci`; para calcular el valor 30 de Fibonacci se requieren 2,692,537 llamadas al método. A medida que trate de calcular números Fibonacci más grandes, observará que cada número Fibonacci consecutivo que pida al applet que calcule, produce un considerable aumento en tiempo de cálculo y en el número de llamadas al método `fibonacci`. Por ejemplo, el valor 31 de Fibonacci requiere 4,356,617 llamadas, y el valor 32 de Fibonacci requiere 7,049,155 llamadas. Como puede ver, el número de llamadas a `fibonacci` se incrementa rápidamente: 1,664,080 llamadas adicionales entre los valores 30 y 31 de Fibonacci, y 2,692,538 llamadas adicionales entre los valores 31 y 32 de Fibonacci. La diferencia entre el número de llamadas realizadas entre los valores 31 y 32 de Fibonacci es de más de 1.5 veces el número de llamadas entre los valores 30 y 31 de Fibonacci. Los problemas de esta naturaleza humillan hasta a las computadoras más poderosas del mundo.[15]

Tip de rendimiento 6.2

Evite los programas recursivos estilo Fibonacci, que producen una "explosión" exponencial de llamadas a métodos.

14. Los lenguajes C y C++ (en los cuales se basan muchas de las características de Java) no especifican el orden en el que se evalúan los operandos de la mayoría de los operadores. En esos lenguajes, los programadores no pueden suponer ninguna información acerca del orden en el que se ejecutan las llamadas en este ejemplo. De hecho, las llamadas podrían ejecutar a `fibonacci( 2 )` primero y a `fibonacci( 1 )` después, o las llamadas podrían ejecutarse en el orden inverso: `fibonacci( 1 )`, seguido de `fibonacci( 2 )`. En este programa y en la mayoría de los otros programas, el resultado final sería el mismo en cualquier caso. Pero en algunos programas, la evaluación de un operando puede llegar a tener efectos secundarios que podrían afectar el resultado final de la operación.

15. En el campo de la teoría de la complejidad, los científicos de computadora estudian qué tan duro trabajan los algoritmos para completar sus tareas. Las cuestiones de complejidad se analizan detalladamente en un curso profesional de ciencias computacionales de nivel superior, generalmente llamado "Algoritmos".

Tip para prevenir errores 6.2

Trate de mejorar el programa de Fibonacci de la figura 6.16, de manera que calcule la cantidad aproximada de tiempo requerido para realizar el cálculo. Para este fin puede llamar al método `currentTimeMillis` de la clase `System`, el cual no toma argumentos y devuelve el tiempo actual de la computadora, en milisegundos. Llame a este método dos veces; una vez antes de la llamada a `fibonacci` y otra vez después de llamar a este método. Guarde cada uno de estos valores y calcule la diferencia en los tiempos para determinar cuántos milisegundos se necesitaron para realizar el cálculo. Muestre su resultado.

6.14 Comparación entre recursividad e iteración

En las secciones anteriores estudiamos los métodos `factorial` y `fibonacci`, que pueden implementarse fácilmente ya sea en forma recursiva o iterativa. En esta sección compararemos las dos metodologías y veremos cuál es la ventaja de que el programador elija una metodología en vez de la otra, en cierta situación particular.

Tanto la iteración como la recursividad se basan en una instrucción de control: en la iteración se utiliza una instrucción de repetición (como `for`, `while` o `do...while`); en la recursividad se utiliza una instrucción de selección (como `if`, `if...else` o `switch`). Tanto la iteración como la recursividad implican la repetición: en la iteración se utiliza explícitamente una instrucción de repetición; en la recursividad la repetición se logra mediante llamadas repetidas a los métodos. Tanto la iteración como la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursividad termina cuando se reconoce un caso base. La iteración con repetición controlada por contador y la recursividad se aproximan poco a poco a la terminación: en la iteración se modifica continuamente un contador hasta que asume un valor que hace que falle la condición de continuación de ciclo; en la recursividad se continúan produciendo versiones más simples del problema original, hasta que se llega al caso base. Tanto la iteración como la recursividad pueden ocurrir infinitamente: un ciclo infinito se produce con la iteración cuando la prueba de continuación de ciclo nunca se vuelve falsa; la recursividad infinita ocurre cuando el paso de recursividad no reduce el problema cada vez en una forma que converja en el caso base.

La recursividad tiene muchos puntos en contra. Invoca el mecanismo repetidamente y, por lo tanto, hay una sobrecarga de llamadas a los métodos. Esta repetición puede ser costosa en términos tanto de tiempo de procesador, como de espacio en memoria. Cada llamada recursiva hace que se cree otra copia del método (en realidad sólo se crean las variables); este conjunto de copias puede consumir considerablemente el espacio en memoria. Sin embargo, la iteración ocurre dentro de un método, por lo que se evitan las llamadas repetidas a los métodos y la asignación de memoria adicional. Entonces ¿para qué elegir la recursividad?

Observación de ingeniería de software 6.8

Cualquier problema que puede resolverse en forma recursiva, también puede resolverse en forma iterativa (no recursiva). Generalmente se prefiere una metodología recursiva a una iterativa cuando la primera refleja el problema con más naturalidad, y se produce un programa más fácil de comprender y depurar. A menudo puede implementarse una metodología recursiva con unas cuantas líneas de código, y la metodología iterativa correspondiente podría requerir de grandes cantidades de código. Otra de las razones para elegir una solución recursiva es que una solución iterativa podría no ser aparente.

Tip de rendimiento 6.3

Evite utilizar la recursividad en situaciones en las que se requiera un buen rendimiento. Las llamadas recursivas toman tiempo y consumen memoria adicional.

Error común de programación 6.16

Hacer que un método no recursivo se llame a sí mismo en forma inadvertida, ya sea directa o indirectamente a través de otro método, puede provocar una recursividad infinita.

La mayoría de los libros de texto de programación hablan de la recursividad mucho después de lo que lo hemos hecho nosotros en este libro. Nosotros pensamos que la recursividad es un tema tan rico y complejo por lo tanto, lo mejor es verlo lo antes posible, para así esparcir ejemplos de este tema en el resto del libro. En la figura 6.18 se sintetizan los ejemplos y ejercicios de recursividad en este texto.

Consideremos nuevamente algunas observaciones que se hacen en forma reiterativa a lo largo de este libro. Es importante tener una buena ingeniería de software. En muchos casos, es importante tener un alto ren-

http://librosysolucionarios.net

Capítulo	Ejemplos y ejercicios de recursividad
6	Método factorial (figura 6.15), Método Fibonacci (figura 6.16), Elevar un entero a una potencia entera (ejercicio 6.36), Torres de Hanoi (ejercicio 6.37), Visualización de la recursividad (ejercicio 6.39), Máximo común divisor (ejercicio 6.40), ¿Qué hace este método? (ejercicio 6.43), Encontrar el error en el método recursivo (ejercicio 6.45).
7	¿Qué hace este programa? (ejercicio 7.16), ¿Qué hace este programa? (ejercicio 7.19), Determinar si una cadena es un palíndromo (ejercicio 7.32), Búsqueda lineal (ejercicio 7.33), Búsqueda binaria (ejercicio 7.34), Ocho reinas (ejercicio 7.35), Imprimir un arreglo (ejercicio 7.36), Imprimir un arreglo en orden inverso (ejercicio 7.37), Valor mínimo en un arreglo (ejercicio 7.38), Quicksort (ejercicio 7.39), Recorrido de laberinto (ejercicio 7.40).
20	Inserción en árbol binario (figura 20.17), Recorrido preorden de un árbol binario (figura 20.17), Recorrido inorden de un árbol binario (figura 20.17), Recorrido postorden de un árbol binario (figura 20.17), Imprimir una lista enlazada en orden inverso (ejercicio 20.20), Búsqueda en una lista enlazada (ejercicio 20.21).

Figura 6.18 Resumen de los ejemplos y ejercicios de recursividad en este texto.

dimiento. Desafortunadamente, estas metas se contraponen entre sí. La buena ingeniería de software es la clave para administrar la tarea de desarrollar sistemas de software más grandes y complejos. El alto rendimiento en estos sistemas es la clave para realizar los sistemas del futuro, que impondrán demandas de cómputo mucho mayores en el hardware. ¿En dónde se acomodan los métodos aquí?

Observación de ingeniería de software 6.9



El proceso de dividir los programas en módulos de una manera ordenada y jerárquica fomenta la buena ingeniería de software, pero tiene un precio.

Tip de rendimiento 6.4



Un programa con muchos módulos realiza un número potencialmente grande de llamadas a los métodos, lo cual consume tiempo de ejecución y espacio en el(los) procesador(es) de una computadora. Las máquinas virtuales de Java de la actualidad son capaces de eliminar la mayor parte de la sobrecarga generada por las llamadas a métodos adicionales de programas con muchos módulos.

Use módulos en sus programas con moderación, teniendo siempre en mente el delicado balance entre el rendimiento y la buena ingeniería de software.

6.15 (Ejemplo práctico opcional) Acerca de los objetos: Cómo identificar las operaciones de las clases

En las secciones "Acerca de los objetos" al final de los capítulos 3, 4 y 5, realizamos los primeros pasos en el diseño orientado a objetos de nuestra simulación de elevador. En el capítulo 3 identificamos las clases que necesitábamos implementar y creamos nuestro primer diagrama de clases. En el capítulo 4 modelamos los atributos de las clases. En el capítulo 5, examinamos el estado de los objetos; además, modelamos las actividades de los objetos y las transiciones de estado. En esta sección nos concentraremos en determinar las *operaciones* (o *comportamientos*) necesarias para implementar la simulación del elevador.

Una operación es un servicio que los objetos de una clase proporcionan a los "clientes" (usuarios) de esos objetos. Considere las operaciones de algunos objetos del mundo real. Las operaciones de un radio incluyen ajustar su estación y volumen (que son invocados generalmente por una persona que ajusta los controles del radio). Las operaciones de un auto incluyen acelerar (operación invocada por el conductor que oprime el pedal del acelerador), desacelerar (operación invocada por el conductor que oprime el pedal del freno o que suelta el pedal del acelerador), voltear y cambiar velocidades.

Podemos derivar muchas de las operaciones de cada clase directamente del enunciado del problema. Para ello, examinamos los verbos y frases verbales en el enunciado. Luego relacionamos cada uno de ellos con las

http://librosysolucionarios.net

Clase	Frases verbales
`Elevador`	Se mueve al otro piso, llega a un piso, restablece el botón del elevador, suena el timbre del elevador, indica su llegada, abre su puerta, cierra su puerta.
`ConductoElevador`	Apaga la luz, enciende la luz, restablece el botón del piso.
`Persona`	Camina en el piso, oprime el botón del piso, oprime el botón del elevador, viaja en el elevador, entra al elevador, sale del elevador.
`Piso`	[Ninguna en el enunciado del problema.]
`BotonPiso`	Solicita el elevador.
`BotonElevador`	Cierra la puerta del elevador, indica al elevador que vaya al piso opuesto.
`PuertaPiso`	Indica a la persona que entre al elevador (al abrirse).
`PuertaElevador`	Indica a la persona que salga del elevador (al abrirse), abre la puerta del piso, cierra la puerta del piso.
`Timbre`	[Ninguna en el enunciado del problema.]
`Luz`	[Ninguna en el enunciado del problema.]

Figura 6.19 Frases verbales para cada clase en el simulador.

clases particulares en nuestro sistema (figura 6.19). Muchas de las frases nominales de la figura 6.19 nos ayudan a determinar las operaciones de nuestras clases.[16]

Para crear operaciones, examinamos las frases verbales enlistadas con cada clase. La frase "se mueve al otro piso" enlistada con la clase `Elevador` se refiere a la actividad en la que el elevador se mueve entre piso y piso. ¿La acción "se mueve" debe ser una operación de la clase `Elevador`? El elevador decide moverse en respuesta a un botón oprimido. Un botón *indica* al elevador que se mueva, pero el botón no es el que *mueve* al elevador; por lo tanto, "se mueve al otro piso" no corresponde a una operación. (Incluiremos las operaciones para instruir al elevador que se mueva al otro piso más adelante, cuando veamos las frases verbales asociadas con los botones.) La frase "llega al piso" tampoco es una operación, ya que el elevador es el que decide cuándo llegar al piso después de viajar cinco segundos.

La frase "restablece el botón del elevador" asociada con la clase `Elevador` implica que el elevador indica al botón del elevador que se restablezca. Por lo tanto, la clase `BotonElevador` necesita una operación para proporcionar este servicio al elevador. Colocamos esta operación (`restablecerBoton`) en el tercer compartimiento de la clase `BotonElevador` en nuestro diagrama de clases (figura 6.20).

Para representar las operaciones enlistamos el nombre de la operación, seguido de una lista de parámetros separados por comas entre paréntesis, y después un signo de dos puntos seguido por el tipo de valor de retorno:

nombre(parámetro1, parámetro2, ..., parámetroN) : tipo de valor de retorno

Por el momento, la mayoría de nuestras operaciones no tienen parámetros. Aún no sabemos los tipos de valores de retorno para muchos de los métodos, por lo que los omitiremos en el diagrama. A medida que avancen nuestros procesos de diseño e implementación, agregaremos las listas de parámetros apropiadas, junto con los tipos de valores de retorno.

De la frase "suena el timbre del elevador" enlistada con la clase `Elevador`, podemos concluir que la clase `Timbre` debe tener una operación que proporcione un servicio (es decir, timbrar). Enlistamos la operación `sonarTimbre` bajo la clase `Timbre`.

Al llegar a un piso, el elevador "indica su llegada" a las puertas. La puerta del elevador responde abriéndose. Por lo tanto, la clase `PuertaElevador` necesita una operación para abrirse. Colocamos la operación `abrirPuerta` en el tercer compartimiento de esta clase. La frase "cierra la puerta [del elevador]" indica que la clase `PuertaElevador` necesita una operación para cerrarse, por lo que colocamos la operación `cerrarPuerta` en el mismo compartimiento.

16. El análisis verbal es un simple medio para identificar comportamientos. En la industria es común utilizar métodos más rigurosos, como los prescritos por el Proceso unificado de Rational (RUP).

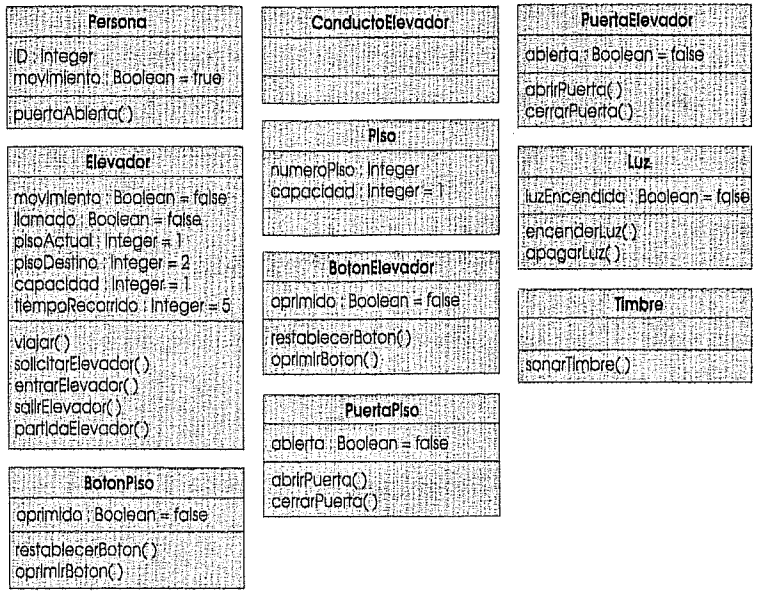


Figura 6.20 Clases con atributos y operaciones.

La clase `ConductoElevador` tiene las frases verbales "apaga la luz" y "enciende la luz", por lo que creamos las operaciones `apagarLuz` y `encenderLuz`, y las enlistamos bajo la clase `Luz`. La frase "restablece el botón del piso" implica que el elevador indica al botón del piso que se restablezca. Por lo tanto, la clase `BotonPiso` necesita una operación `restablecerBoton`.

La frase "camina en el piso" enlistada por la clase `Persona` no es una operación, ya que una persona decide caminar por el piso por su propia iniciativa. Sin embargo, las frases "oprime el botón del piso" y "oprime el botón del elevador" son operaciones que corresponden a las clases de los botones. Por lo tanto, colocamos la operación `oprimirBoton` bajo las clases `BotonPiso` y `BotonElevador` en nuestro diagrama de clases (figura 6.20). La frase "viaja en el elevador" implica que `Elevador` necesita una operación que permita a una persona viajar en el elevador, por lo que colocamos la operación `viajar` en el compartimiento inferior de `Elevador`. Las frases "entra al elevador" y "sale del elevador", enlistadas con la clase `Persona`, sugieren que la clase `Elevador` necesita operaciones que correspondan a estas acciones.[17] Colocamos las operaciones `entrarElevador` y `salirElevador` en el compartimiento inferior de la clase `Elevador`.

La frase "solicita el elevador" enlistada bajo la clase `BotonPiso` implica que la clase `Elevador` necesita una operación `solicitarElevador`. La frase "indica al elevador que vaya al piso opuesto" enlistada con la clase `BotonElevador` implica que `BotonElevador` indica a `Elevador` que se vaya. Por lo tanto, el `Elevador` necesita proporcionar un servicio de "partida"; colocamos una operación `partidaElevador` en el compartimiento inferior de `Elevador`.

Las frases enlistadas con las clases `PuertaPiso` y `PuertaElevador` mencionan que las puertas (al abrirse) indican a un objeto `Persona` que entre o salga del elevador. Específicamente, una puerta informa a una persona que la puerta se ha abierto. (La persona entonces entra o sale del elevador.) Colocamos la operación `puertaAbierta` en el compartimiento inferior para la clase `Persona`. Además, la `PuertaElevador` abre y cierra la `PuertaPiso`, por lo que asignamos `abrirPuerta` y `cerrarPuerta` al compartimiento inferior de la clase `PuertaPiso`.

17. En este punto sólo podemos adivinar lo que estas operaciones hacen. Por ejemplo, tal vez estas operaciones modelen los elevadores reales, algunos de los cuales tienen sensores que detectan cuándo entran y salen los pasajeros. Por ahora, simplemente enlistaremos estas operaciones. Más adelante descubriremos, en todo caso, qué acciones realizan estas operaciones a medida que avancemos por nuestro proceso de diseño.

Por ahora no tenemos que preocuparnos con los parámetros de las operaciones o los tipos de valores de retorno; trataremos de obtener sólo una comprensión básica de las operaciones de cada clase. A medida que continuemos por nuestro proceso de diseño, el número de operaciones que pertenezcan a cada clase podrá variar (tal vez descubramos que se necesitan nuevas operaciones, o que algunas operaciones actuales son innecesarias), y podríamos determinar que algunas de las operaciones de nuestras clases necesitan ciertos tipos de valores de retorno específicos.

RESUMEN

- La mejor manera de desarrollar y mantener un programa extenso es dividirlo en varios módulos pequeños. En Java los módulos se escriben como clases y métodos.
- Un método es una unidad ejecutable, invocada por la llamada a ese método. La llamada al método menciona su nombre y proporciona argumentos entre paréntesis, los cuales son requeridos por el método llamado para realizar su tarea. Si el método está siendo llamado a un objeto, la llamada debe ir precedida por el nombre de una referencia y un punto. Si el método es `static`, debe anteponerse el nombre de una clase y un punto.
- Cada uno de los argumentos de un método puede ser una constante, una variable o una expresión.
- Una variable local se conoce solamente en el método que declara a esa variable. No se permite a los métodos conocer los detalles de implementación de ningún otro método.
- El área visual en pantalla para un objeto de la clase `JApplet` tiene un panel de contenido, al cual deben adjuntarse los componentes de la GUI, de manera que puedan mostrarse en tiempo de ejecución. El panel de contenido es un objeto de la clase `Container` que se encuentra en el paquete `java.awt`. El método `getContentPane` de `JApplet` devuelve una referencia al panel de contenido del applet.
- El formato básico de la declaración de un método es

  tipo-de-valor-de-retorno nombre-del-método(parámetro1, parámetro2, ..., parámetroN)
  {
  declaraciones e instrucciones
  }

  El *tipo-de-valor-de-retorno* especifica el tipo de valor que se devuelve al método que hizo la llamada. Si un método no devuelve un valor, entonces su *tipo-de-valor-de-retorno* es `void`. El *nombre-del-método* puede ser cualquier identificador válido. Los *parámetros* se declaran en una lista separada por comas (la lista de parámetros) encerrada entre paréntesis, la cual declara el tipo y nombre de cada parámetro. Si un método no acepta ningún valor, la lista de parámetros debe estar vacía. El cuerpo del método es un bloque de *declaraciones e instrucciones* que constituyen a ese método.
- Los argumentos que se pasan a un método deben concordar en número, tipo y orden con los parámetros en la declaración del método.
- Cuando un programa encuentra la llamada a un método, el control se transfiere desde el punto de la invocación hasta el método llamado, el método se ejecuta y el control regresa al método que hizo la llamada. Un método llamado puede devolver el control al solicitante en una de tres formas. Si un método tiene un tipo de valor de retorno `void`, el control regresa a la llave derecha de terminación del método, o mediante la ejecución de la instrucción

  `return;`

  Si el método devuelve un valor, la instrucción

  `return` *expresión*`;`

  devuelve el valor de la *expresión*.
- Existen tres formas de llamar a un método: usando el nombre del método por sí solo; usando una referencia a un objeto, seguida de un punto (`.`) y el nombre del método; y usando el nombre de una clase, seguido de un punto (`.`) y el nombre del método. La última sintaxis es para métodos `static`.

http://librosysolucionarios.net

- En muchos casos, los valores de los argumentos que no corresponden precisamente a los tipos de los parámetros en la declaración del método se convierten al tipo apropiado antes de llamar al método. Esto puede provocar errores de compilación, si se quebrantan las reglas de promoción de Java.
- El método `Math.random` genera un valor `double` de 0.0 hasta, pero sin incluir a, 1.0. Los valores producidos por `Math.random` pueden escalarse y desplazarse para producir valores en un rango dado.
- La ecuación general para escalar y desplazar un número aleatorio es

  `n` = *valorDesplazamiento* `+ ( int ) ( Math.random() *` *factorEscala* `);`

- Una clase puede heredar los atributos y comportamientos (campos y métodos) existentes de otra clase que se especifique a la derecha de la palabra clave `extends` en la declaración de la clase. Además, una clase puede implementar una o más interfaces. Una interfaz especifica uno o más métodos que se deben declarar en la clase implementadora.
- La interfaz `ActionListener` especifica que una clase debe implementar a un método con la primera línea que se muestra a continuación:

```
public void actionPerformed( ActionEvent actionEvent )
```

- El método `actionPerformed` procesa la interacción de un usuario con un componente de la GUI que genera un evento de acción. El método se llama en respuesta a la interacción del usuario. A este proceso se le conoce como manejo de eventos. El método `actionPerformed` es un manejador de eventos. Este estilo de programación se conoce como programación controlada por eventos.
- La palabra clave `final` declara variables constantes, que deben inicializarse una vez antes de utilizarse en un programa. Las variables constantes también se conocen como constantes, constantes con nombre o variables de sólo lectura.
- Un objeto de la clase `JLabel` contiene una cadena de caracteres que se van a mostrar en la pantalla. Por lo general, un objeto `JLabel` indica el propósito de otro elemento de la GUI en la pantalla.
- Los objetos de la clase `JTextField` obtienen información del usuario o muestran información en la pantalla.
- Cuando el usuario oprime un objeto `JButton`, el programa generalmente responde realizando una tarea.
- El método `setLayout` de la clase `Container` especifica el administrador de esquemas para la interfaz de usuario del applet. Los administradores de esquemas se proporcionan para ordenar los componentes de la GUI en un objeto `Container`, para presentarlos en la pantalla. En un esquema `FlowLayout`, los componentes de la GUI se colocan en un objeto `Container` de izquierda a derecha en el orden en el que se adjuntaron al objeto `Container` con el método `add`. Cuando se llega al borde del contenedor, los componentes deben continuarse en la siguiente línea.
- Antes de poder procesar cualquier evento, cada componente de la GUI debe saber cuál es el objeto en el programa que contiene el método manejador de eventos que se llamará cuando ocurra un evento. El método `addActionListener` se utiliza para registrar un evento para un objeto `JButton` o `JTextField`. Para responder a un evento de acción, una clase debe implementar a `ActionListener` y al método `actionPerformed`.
- El método `showStatus` muestra una cadena en la barra de estado del contenedor de applets.
- El alcance de una declaración es la porción de un programa que puede hacer referencia a la entidad en la declaración, por su nombre.
- El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece la declaración.
- El alcance de la declaración de una variable local es desde el punto en el que aparece la declaración en el bloque, hasta el final de ese bloque.
- El alcance de una etiqueta en una instrucción `break` o `continue` etiquetada es la instrucción encerrada por la instrucción etiquetada (es decir, el cuerpo de la instrucción etiquetada).
- El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for` y las demás expresiones en el encabezado.
- El alcance de un método o campo de una clase es todo el cuerpo de la clase. Esto permite a los métodos de una clase utilizar nombres simples para llamar a los demás métodos declarados en la clase o heredados por esa clase, y les permite también utilizar los campos declarados en la clase.
- Un método recursivo es un método que se llama a sí mismo, ya sea directa o indirectamente.
- Si un método recursivo es llamado con un caso base, el método devuelve inmediatamente un resultado. Si el método es llamado con un problema más complejo, el método divide el problema en dos o más piezas conceptuales: una pieza que

el método sabe cómo realizar, y una versión ligeramente más pequeña del problema original. Como este nuevo problema se ve igual que el problema original, el método hace una llamada recursiva para trabajar en el problema más pequeño.

- Para que la recursividad termine, la secuencia de problemas más y más pequeños debe convergir en el caso base. Cuando el método reconoce el caso base, el resultado se devuelve a la llamada anterior al método, y se origina una secuencia de resultados de vuelta hasta la llamada original, hasta que ésta devuelve el resultado original.
- El método `init` de un applet se llama una vez por el contenedor de applets, cuando se carga un applet para ejecutarse. Este método lleva a cabo la inicialización de un applet. El método `start` del applet se llama después de que el método `init` termina de ejecutarse, y cada vez que el usuario del navegador regresa a la página de HTML en la que reside el applet (después de navegar en otra página de HTML).
- El método `paint` de un applet se llama después de que el método `start` ha empezado a ejecutarse, para dibujar en el applet. También se llama cada vez que el applet necesita volver a pintarse.
- El método `stop` de un applet se llama cuando el applet debe suspender su ejecución; generalmente, cuando el usuario del navegador sale de la página de HTML en la que reside el applet.
- El método `destroy` de un applet se llama cuando el applet se va a quitar de la memoria; generalmente, cuando el usuario del navegador sale de la sesión de navegación.
- El método `repaint` puede llamarse en un applet para ocasionar una llamada nueva a `paint`. El método `repaint` llama a otro método, de nombre `update`, y le pasa el objeto `Graphics`. El método `update` llama al método `paint` y le pasa el objeto `Graphics`.
- La sobrecarga de métodos ocurre cuando los métodos tienen el mismo nombre, pero distintas listas de parámetros. Cuando se realiza una llamada a un método sobrecargado, el compilador selecciona el método apropiado examinando los argumentos en la llamada. Los métodos sobrecargados deben tener listas de parámetros distintas.

TERMINOLOGÍA

`ActionEvent`, clase
`ActionListener`, interfaz
`ActionPerformed`, método de `ActionListener`
alcance
API de Java (biblioteca de clases de Java)
argumento en la llamada a un método
bloque
caso base en la recursividad
clase
coerción de argumentos
constante
constante con nombre
copia de un valor
declaración de un método
desplazar
`destroy`, método de la clase `JApplet`
devolver
divide y vencerás
elemento de azar
escalar
expresión de tipo mixto
`final`
firma
firma de un método
`FlowLayout`, clase
generación de números aleatorios
ingeniería de software
`init`, método de la clase `JApplet`
interfaz
invocar a un método
iteración
`JButton`, clase
`JLabel`, clase
`JTextField`
llamada a un método
llamada recursiva
llamar a un método
`Math`, clase
`Math.E`
`Math.PI`
`Math.random`, método
método
método definido por el programador
método factorial
método llamado
método que hace la llamada (solicitante)
método recursivo
`paint`, método de la clase `JApplet`
parámetro en la declaración de un método
paso de recursividad
programa modular
programación controlada por eventos
recursividad
reglas de promoción
`repaint`, método de la clase `JApplet`
reutilización de software
`setLayout`, método de la clase `JApplet`
`showStatus`, método de la clase `JApplet`
simulación
sobrecarga de un método
`start`, método de la clase `JApplet`
`stop`, método de la clase `JApplet`
`this`
tipo de valor de retorno
tipos de referencias
`update`, método de la clase `JApplet`
variable constante
variable de sólo lectura
variable local
`void`

http://librosysolucionarios.net

EJERCICIOS DE AUTOEVALUACIÓN

6.1 Complete las siguientes oraciones:

a) Un método se invoca con un(a) _____.

b) A una variable que se conoce sólo dentro del método en el que está declarada, se le llama _____.

c) La instrucción _____ en un método llamado puede usarse para pasar el valor de una expresión, de vuelta al método que hizo la llamada.

d) La palabra clave _____ indica que un método no devuelve ningún valor.

e) El _____ de una declaración es la porción de un programa que puede hacer referencia a la entidad en la declaración, por su nombre.

f) Las tres formas de regresar el control de un método llamado a un solicitante son _____, _____ y _____.

g) El método _____ se llama una vez cuando un applet empieza a ejecutarse.

h) El método _____ produce números aleatorios.

i) El método _____ se llama cada vez que el usuario de un navegador vuelve a visitar la página de HTML en la que reside el applet.

j) El método _____ se invoca para dibujar en un applet.

k) El método _____ invoca al método `update` de un applet, el cual a su vez invoca al método `paint` del applet.

l) El método _____ se invoca para un applet cada vez que el usuario de un navegador sale de una página de HTML en la que reside el applet.

m) Un método que se llama a sí mismo, ya sea directa o indirectamente, es un método _____.

n) Un método recursivo generalmente tiene dos componentes: uno que proporciona el medio para que termine la recursividad, evaluando un caso _____, y uno que expresa el problema como una llamada recursiva para un problema ligeramente más simple que el de la llamada original.

o) En Java, es posible tener varios métodos con el mismo nombre, en los que cada uno opera distintos tipos o números de argumentos. Esta característica se conoce como _____ de métodos.

p) El modificador _____ se utiliza para declarar variables constantes.

6.2 Para el siguiente programa, indique el alcance de cada una de las siguientes entidades:

a) La variable `x`.

b) La variable `y`.

c) El método `cubo`.

d) El método `paint`.

e) La variable `yPos`.

```
1  public class PruebaCubo extends JApplet {
2     int x;
3
4     public void paint( Graphics g )
5     {
6        int yPos = 25;
7
8        for ( x = 1; x <= 10; x++ ) {
9           g.drawString( cubo( x ), 25, yPos );
10          yPos +=15;
11       }
12    }
13
14    public int cubo( int y )
15    {
16       return y * y * y;
17    }
18 }
```

6.3 Escriba una aplicación que pruebe si los ejemplos de las llamadas a los métodos de la biblioteca de matemáticas mostradas en la figura 6.2 realmente producen los resultados indicados.

6.4 Proporcione el encabezado para cada uno de los siguientes métodos:

http://librosysolucionarios.net

a) El método `hipotenusa`, que toma dos argumentos de punto flotante con doble precisión, llamados `lado1` y `lado2`, y que devuelve un resultado de punto flotante, con doble precisión.

b) El método `menor`, que toma tres enteros `x`, `y` y `z`, y devuelve un entero.

c) El método `instrucciones`, que no toma argumentos y no devuelve ningún valor. (*Nota*: Estos métodos se utilizan comúnmente para mostrar instrucciones a un usuario.)

d) El método `intAFloat`, que toma un argumento entero llamado `numero` y devuelve un resultado de punto flotante.

6.5 Encuentre el error en cada uno de los siguientes segmentos de programas. Explique cómo se puede corregir el error.

a)
```
int g() {
   System.out.println( "Dentro del método g" );
   int h() {
      System.out.println( "Dentro del método h");
   }
}
```

b)
```
int suma( int x, int y ) {
   int resultado;
   resultado = x + y;
}
```

c)
```
int suma( int n ) {
   if ( n == 0 )
      return 0;
   else
      n + suma( n - 1 );
}
```

d)
```
void f( float a ); {
   float a;
   System.out.println( a );
}
```

e)
```
void producto() {
   int a = 6, b = 5, c = 4, resultado;
   resultado = a * b * c;
   System.out.println( "El resultado es " + resultado );
   return resultado;
}
```

6.6 Escriba un applet completo en Java que pida al usuario el radio tipo `double` de una esfera, y que llame al método `volumenEsfera` para calcular y mostrar el volumen de esa esfera, utilizando la asignación

```
volumen = ( 4.0 / 3.0 ) * Math.PI * Math.pow ( radio, 3 )
```

El usuario debe introducir el radio mediante un objeto `JTextField`.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

6.1 a) Llamada a un método. b) Variable local. c) `return`. d) `void`. e) Alcance. f) `return;` `return` *expresión*; o encontrar la llave derecha de cierre de un método. g) `init`. h) `Math.random`. i) `start`. j) `paint`. k) `repaint`. l) `stop`. m) Recursivo. n) Base. o) Sobrecarga. p) `final`.

6.2 a) El cuerpo de la clase. b) El bloque que define el cuerpo del método `cubo`. c) El cuerpo de la clase. d) El cuerpo de la clase. e) El bloque que define el cuerpo del método `paint`.

6.3 La siguiente solución demuestra el uso de los métodos de la clase `Math` de la figura 6.2:

```
1  // Ejercicio 6.3: PruebaMath.java
2  // Prueba de los métodos de la clase Math.
3
4  public class PruebaMath {
5
6     public static void main( String args[] )
7     {
8        System.out.println( "Math.abs( 23.7 ) = " + Math.abs( 23.7 ) );
```

```
         System.out.println( "Math.abs( 0.0 ) = " + Math.abs( 0.0 ) );
         System.out.println( "Math.abs( -23.7 ) = " + Math.abs( -23.7 ) );
         System.out.println( "Math.ceil( 9.2 ) = " + Math.ceil( 9.2 ) );
         System.out.println( "Math.ceil( -9.8 ) = " + Math.ceil( -9.8 ) );
         System.out.println( "Math.cos( 0.0 ) = " + Math.cos( 0.0 ) );
         System.out.println( "Math.exp( 1.0 ) = " + Math.exp( 1.0 ) );
         System.out.println( "Math.exp( 2.0 ) = " + Math.exp( 2.0 ) );
         System.out.println( "Math.floor( 9.2 ) = " + Math.floor( 9.2 ) );
         System.out.println( "Math.floor( -9.8 ) = " + Math.floor( -9.8 ) );
         System.out.println( "Math.log( Math.E ) = " +
            Math.log( Math.E ) );
         System.out.println( "Math.log( Math.E * Math.E ) = " +
            Math.log( Math.E * Math.E ) );
         System.out.println( "Math.max( 2.3, 12.7 ) = " +
            Math.max( 2.3, 12.7 ) );
         System.out.println( "Math.max( -2.3, -12.7 ) = " +
            Math.max( -2.3, -12.7 ) );
         System.out.println( "Math.min( 2.3, 12.7 ) = " +
            Math.min( 2.3, 12.7 ) );
         System.out.println( "Math.min( -2.3, -12.7 ) = " +
            Math.min( -2.3, -12.7 ) );
         System.out.println( "Math.pow( 2.0, 7.0 ) = " +
            Math.pow( 2.0, 7.0 ) );
         System.out.println( "Math.pow( 9.0, 0.5 ) = " +
            Math.pow( 9.0, 0.5 ) );
         System.out.println( "Math.sin( 0.0 ) = " + Math.sin( 0.0 ) );
         System.out.println( "Math.sqrt( 900.0 ) = " + Math.sqrt( 900.0 ) );
         System.out.println( "Math.sqrt( 9.0 ) = " + Math.sqrt( 9.0 ) );
         System.out.println( "Math.tan( 0.0 ) = " + Math.tan( 0.0 ) );

      } // fin de main

} // fin de la clase PruebaMath
```

```
Math.abs( 23.7 ) = 23.7
Math.abs( 0.0 ) = 0.0
Math.abs( -23.7 ) = 23.7
Math.ceil( 9.2 ) = 10.0
Math.ceil( -9.8 ) = -9.0
Math.cos( 0.0 ) = 1.0
Math.exp( 1.0 ) = 2.7182818284590455
Math.exp( 2.0 ) = 7.38905609893065
Math.floor( 9.2 ) = 9.0
Math.floor( -9.8 ) = -10.0
Math.log( Math.E ) = 1.0
Math.log( Math.E * Math.E ) = 2.0
Math.max( 2.3, 12.7 ) = 12.7
Math.max( -2.3, -12.7 ) = -2.3
Math.min( 2.3, 12.7 ) = 2.3
Math.min( -2.3, -12.7 ) = -12.7
Math.pow( 2.0, 7.0 ) = 128.0
Math.pow( 9.0, 0.5 ) = 3.0
Math.sin( 0.0 ) = 0.0
Math.sqrt( 900.0 ) = 30.0
Math.sqrt( 9.0 ) = 3.0
Math.tan( 0.0 ) = 0.0
```

http://librosysolucionarios.net

6.4 a) `double hipotenusa( double lado1, double lado2 )`
b) `int menor( int x, int y, int z )`
c) `void instrucciones()`
d) `float intoFloat( int numero )`

6.5 a) Error: el método `h` está declarado dentro del método `g`.
Corrección: mueva la declaración de `h` fuera de la declaración de `g`.
b) Error: se supone que el método debe devolver un entero, pero no es así.
Corrección: elimine la variable `resultado`, y coloque la instrucción

```
return x + y;
```

en el método, o agregue la siguiente instrucción al final del cuerpo del método:

```
return resultado;
```

c) Error: este método recursivo no devuelve el resultado de `n + suma( n - 1 )`, lo que produce un error de sintaxis.
Corrección: modifique la instrucción en la cláusula `else` para que quede como

```
return n + suma( n - 1 );
```

d) Error: el punto y coma que va después del paréntesis de la lista de parámetros está incorrecto y el parámetro `a` no debe volver a declararse en el método.
Corrección: elimine el punto y coma que va después del paréntesis derecho de la lista de parámetros, y elimine la declaración `float a;`.
e) Error: el método devuelve un valor cuando no debe hacerlo.
Corrección: cambie el tipo de valor de retorno de `void` a `int`.

6.6 La siguiente solución calcula el volumen de una esfera, utilizando el radio introducido por el usuario:

```
// Ejercicio 6.6: PruebaEsfera.java
// Calcula el volumen de una esfera.
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class PruebaEsfera extends JApplet implements ActionListener {
   JLabel indicadorEtiqueta;
   JTextField entradaCampo;

   // crear GUI
   public void init()
   {
      Container contenedor = getContentPane();
      contenedor.setLayout( new FlowLayout() );

      indicadorEtiqueta = new JLabel( "Escriba el radio de la esfera: " );
      entradaCampo = new JTextField( 10 );
      entradaCampo.addActionListener( this );
      contenedor.add( indicadorEtiqueta );
      contenedor.add( entradaCampo );

   } // fin del método init

   // calcular el volumen de la esfera cuando el usuario oprima Intro en
      entradaCampo
   public void actionPerformed( ActionEvent actionEvent )
   {
      double radio =
         Double.parseDouble( actionEvent.getActionCommand() );

      showStatus( "El volumen es " + volumenEsfera( radio ) );
```

```
33
34     } // fin del método actionPerformed
35
36     // calcular y devolver el volumen de la esfera
37     public double volumenEsfera( double radio )
38     {
39        double volumen = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radio, 3 );
40
41        return volumen;
42
43     } // fin del método esferaVolumen
44
45  } // fin de la clase PruebaEsfera
```



EJERCICIOS

6.7 ¿Cuál es el valor de `x` después de que se ejecuta cada una de las siguientes instrucciones?

a) `x = Math.abs( 7.5 );`
b) `x = Math.floor( 7.5 );`
c) `x = Math.abs( 0.0 );`
d) `x = Math.ceil( 0.0 );`
e) `x = Math.abs( -6.4 );`
f) `x = Math.ceil( -6.4 );`
g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

6.8 Un estacionamiento cobra una cuota mínima de $2.00 por estacionarse hasta tres horas. El estacionamiento cobra $0.50 adicionales por cada hora o *fracción* que se pase de tres horas. El cargo máximo para cualquier periodo dado de 24 horas es de $10.00. Suponga que ningún auto se estaciona durante más de 24 horas a la vez. Escriba un applet que calcule y muestre los cargos por estacionamiento para cada cliente que se haya estacionado ayer. Debe introducir en un objeto `JTextField` las horas de estacionamiento para cada cliente. El programa debe mostrar el cargo para el cliente actual y debe calcular y mostrar el total corriente de los recibos de ayer. El programa debe utilizar el método `calcularCargos` para determinar el cargo para cada cliente.

6.9 Una aplicación del método `Math.floor` es redondear un valor al siguiente entero. La instrucción

```
y = Math.floor( x + 0.5 );
```

redondea el número `x` al entero más cercano y asigna el resultado a `y`. Escriba un applet que lea valores `double` y que utilice la instrucción anterior para redondear cada uno de los números a su entero más cercano. Para cada número procesado, muestre tanto el número original como el redondeado.

6.10 `Math.floor` puede utilizarse para redondear un número hasta un lugar decimal específico. La instrucción

```
y = Math.floor( x * 10 + 0.5 ) / 10;
```

redondea `x` en la posición de las décimas (es decir, la primera posición a la derecha del punto decimal). La instrucción

```
y = Math.floor( x * 100 + 0.5 ) / 100;
```

redondea `x` en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba un applet que defina cuatro métodos para redondear un número `x` en varias formas:

a) `redondearAInteger( numero )`
b) `redondearADecimas( numero )`
c) `redondearACentesimas( numero )`
d) `redondearAMilesimas( numero )`

Para cada valor leído, su programa debe mostrar el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana y el número redondeado a la milésima más cercana.

6.11 Responda a cada una de las siguientes preguntas:

a) ¿Qué significa elegir números "al azar"?
b) ¿Por qué es el método `Math.random` útil para simular juegos al azar?
c) ¿Por qué es a menudo necesario escalar o desplazar los valores producidos por `Math.random`?
d) ¿Por qué es la simulación computarizada de las situaciones reales una técnica útil?

6.12 Escriba instrucciones que asignen enteros aleatorios a la variable n en los siguientes rangos:

a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$
e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

6.13 Para cada uno de los siguientes conjuntos de enteros, escriba una sola instrucción que imprima un número al azar del conjunto:

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

6.14 Escriba un método llamado `enteroPotencia( base, exponente )` que devuelva el valor de

$base^{exponente}$

Por ejemplo, `enteroPotencia( 3, 4 )` calcula 3^4 (o `3 * 3 * 3 * 3`). Suponga que `exponente` es un entero positivo distinto de cero y que `base` es un entero. El método `enteroPotencia` debe utilizar un ciclo `for` o `while` para controlar el cálculo. No utilice ningún método de la biblioteca de matemáticas. Incorpore este método en un applet que lea valores enteros para `base` y `exponente` mediante objetos `JTextField`, y que realice el cálculo con el método `enteroPotencia`. (*Nota:* Registre el manejo de eventos sólo para el segundo objeto `JTextField`. El usuario debe interactuar con el programa escribiendo números en ambos objetos `JTextField`, pero oprimiendo *Intro* solamente en el segundo objeto `JTextField`.)

6.15 Defina un método llamado `hipotenusa` que calcule la longitud de la hipotenusa de un triángulo recto, cuando se proporcionen las longitudes de los otros dos lados. (Utilice los datos de ejemplo de la figura 6.21.) El método debe tomar dos argumentos de tipo `double` y devolver la hipotenusa como un valor `double`. Incorpore este método en un applet que lea los valores para `lado1` y `lado2` mediante objetos `JTextField`, y que realice el cálculo con el método `hipotenusa`. Determine la longitud de la hipotenusa para cada uno de los triángulos de la figura 6.21. (*Nota:* Registre el manejo de eventos sólo para el segundo objeto `JTextField`. El usuario debe interactuar con el programa escribiendo números en ambos objetos `JTextField`, pero oprimiendo *Intro* solamente en el segundo objeto `JTextField`.)

6.16 Escriba un método llamado `multiple` que determine, para un par de enteros, si el segundo entero es múltiplo del primero. El método debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero, y `false` en caso contrario. Incorpore este método en un applet que reciba como entrada una serie de pares de en-

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Figura 6.21 Valores para los lados de los triángulos del ejercicio 6.15.

http://librosysolucionarios.net

teros (un par a la vez, usando objetos `JTextField`). (*Nota:* Registre el manejo de eventos sólo para el segundo objeto `JTextField`. El usuario debe interactuar con el programa escribiendo números en ambos objetos `JTextField`, pero oprimiendo *Intro* solamente en el segundo objeto `JTextField`.)

6.17 Escriba un método llamado `esPar` que utilice el operador residuo (`%`) para determinar si un entero dado es par. El método debe tomar un argumento entero y devolver `true` si el entero es par, y `false` en caso contrario. Incorpore este método en un applet que reciba como entrada una secuencia de enteros (uno a la vez, utilizando un objeto `JTextField`).

6.18 Escriba un método llamado `cuadradoDeAsteriscos` que muestre un cuadrado relleno (el mismo número de filas y columnas) de asteriscos cuyo lado sea especificado en el parámetro entero `lado`. Por ejemplo, si `lado` es `4`, el método debe mostrar el patrón de asteriscos que se muestra a continuación.

```
****
****
****
****
```

Incorpore este método en un applet que reciba como entrada por parte del usuario un valor entero para `lado`, y que realice el dibujo con el método `cuadradoDeAsteriscos`. Observe que este método debe llamarse desde el método `paint` del applet, y que debe recibir el objeto `Graphics` de `paint`.

6.19 Modifique el método creado en el ejercicio 6.18 para formar el cuadrado de cualquier carácter que esté contenido en el parámetro tipo carácter `caracterRelleno`. Por ejemplo, si `lado` es `5` y `caracterRelleno` es "#", el método debe imprimir

```
#####
#####
#####
#####
#####
```

6.20 Escriba un applet que utilice un método llamado `circuloArea`, que pida al usuario el radio de un círculo y que calcule e imprima el área de ese círculo.

6.21 Modifique el programa del ejercicio 6.18 para dibujar un cuadrado relleno con el método `fillRect` de la clase `Graphics`. El método `fillRect` requiere dos argumentos; coordenada *x*, coordenada *y*, anchura y altura. Usted debe permitir que el usuario introduzca las coordenadas en las que debe aparecer el cuadrado, y la longitud de uno de sus lados.

6.22 Escriba segmentos de programas que realicen cada una de las siguientes tareas:

a) Calcular la parte entera del cociente, cuando el entero `a` se divide entre el entero `b`.

b) Calcular el residuo entero cuando el entero `a` se divide entre el entero `b`.

c) Utilizar las piezas de los programas desarrollados en las partes (a) y (b) para escribir un método llamado `mostrarDigitos`, que reciba un entero entre `1` y `99999`, y que lo muestre como una secuencia de dígitos, separando cada par de dígitos por dos espacios. Por ejemplo, el entero `4562` debe aparecer como

```
4  5  6  2
```

d) Incorpore el método desarrollado en la parte (c) en un applet que reciba como entrada un entero mediante un campo de texto, y que llame al método `mostrarDigitos` pasándole a este método el entero introducido. Muestre los resultados en un segundo campo de texto.

6.23 Implemente los siguientes métodos enteros:

a) El método `centigrados` que devuelve la equivalencia en grados Centígrados de una temperatura en grados Fahrenheit, utilizando el cálculo

```
C = 5.0 / 9.0 * ( F - 32 );
```

b) El método `fahrenheit` que devuelve la equivalencia en grados Fahrenheit de una temperatura en grados Centígrados, utilizando el cálculo

```
F = 9.0 / 5.0 * C + 32;
```

c) Utilice los métodos de las partes (a) y (b) para escribir un applet que permite al usuario, ya sea escribir una temperatura en grados Fahrenheit y mostrar su equivalente en grados Centígrados, o escribir una temperatura en grados Centígrados y mostrar su equivalente en grados Fahrenheit.

[*Nota:* Este applet requerirá dos objetos `JTextField` que tengan eventos de acción registrados. Cuando se llama a `actionPerformed`, el parámetro `ActionEvent` cuenta con el método `getSource()` para determinar con cuál componente de la GUI interactuó el usuario. Su método `actionPerformed` debe contener una instrucción `if...else` de la forma

```
if ( actionEvent.getsource() == entrada1 ) {
   // procesar la interacción de entrada1 aquí
}
else {  // actionEvent.getsource() == entrada2
   // procesar la interacción de entrada2 aquí
}
```

en donde `entrada1` y `entrada2` son referencias a objetos `JTextField`.]

6.24 Escriba un método llamado `minimo3` que devuelva el menor de tres números de punto flotante. Use el método `Math.min` para implementar `minimo3`. Incorpore el método en un applet que reciba como entrada los valores por parte del usuario, y determine el valor menor. Muestre el resultado en la barra de estado.

6.25 Se dice que un número entero es un *número perfecto* si sus factores, incluyendo 1 (pero no el número entero), al sumarse dan como resultado el número entero. Por ejemplo, 6 es un número perfecto ya que 6 = 1 + 2 + 3. Escriba un método llamado `perfecto` que determine si el parámetro `numero` es un número perfecto. Use este método en un applet que determine y muestre todos los números perfectos entre 1 y 1000. Imprima los factores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora, evaluando números más grandes que 1000. Muestre los resultados en un objeto `JTextArea` que tenga funcionalidad de desplazamiento.

6.26 Se dice que un entero es *primo* si puede dividirse solamente por 1 y por sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no.

a) Escriba un método que determine si un número es primo.

b) Use este método en un applet que determine e imprima todos los números primos menores que 10,000. ¿Cuántos números hasta 10,000 tiene que probar para asegurarse de encontrar todos los primos? Muestre los resultados en un objeto `JTextArea` que tenga funcionalidad de desplazamiento.

c) Al principio podría pensarse que $n/2$ es el límite superior para evaluar si un número es primo, pero lo máximo que se necesita es hasta la raíz cuadrada de *n*. ¿Por qué? Vuelva a escribir el programa y ejecútelo de ambas formas. Haga una estimación de la mejora en rendimiento.

6.27 Escriba un método que tome un valor entero y devuelva el número con sus dígitos invertidos. Por ejemplo, para el número 7631, el método debe regresar 1367. Incorpore el método en un applet que reciba como entrada un valor del usuario. Muestre el resultado de ese método en la barra de estado.

6.28 El *máximo común divisor (MCD)* de dos enteros es el entero más grande que puede dividir a cada uno de los dos números. Escriba un método llamado `mcd` que devuelva el máximo común divisor de dos enteros. Incorpore el método en un applet que reciba como entrada dos valores del usuario. Muestre el resultado del método en la barra de estado.

6.29 Escriba un método llamado `puntosCalidad` que reciba como entrada el promedio de un estudiante y devuelva `4` si el promedio se encuentra entre 90 y 100, `3` si el promedio se encuentra entre 80 y 89, `2` si el promedio se encuentra entre 70 y 79, `1` si el promedio se encuentra entre 60 y 69, y `0` si el promedio es menor de 60. Incorpore el método en un applet que reciba como entrada un valor del usuario. Muestre el resultado del método en la barra de desplazamiento.

6.30 Escriba un applet que simule el lanzamiento de monedas. Deje que el programa lance una moneda cada vez que el usuario oprima el botón `"Lanzar"`. Cuente el número de veces que aparezca cada uno de los lados de la moneda. Muestre los resultados. El programa debe llamar a un método separado, llamado `tirar`, que no tome argumentos y devuelva `false` en caso de cara, y `true` en caso de cruz. (*Nota:* Si desea que el programa simule en forma realista el lanzamiento de monedas, cada lado de la moneda debe aparecer aproximadamente la mitad del tiempo.)

6.31 Las computadoras están tomando un papel cada vez más importante en la educación. Escriba un programa que ayude a un estudiante de escuela primaria, para que aprenda a multiplicar. Use el método `Math.random` para producir dos enteros positivos de un dígito. El programa debe entonces mostrar una pregunta en la barra de estado, como

```
¿Cuánto es 6 por 7?
```

El estudiante entonces debe escribir la respuesta en un objeto `JTextField`. Luego, el programa debe verificar la respuesta del estudiante. Si es correcta, dibuje la cadena `"¡Muy bien!"` en el applet y haga otra pregunta de multiplicación. Si la respuesta es incorrecta, dibuje la cadena `"No. Por favor intenta de nuevo."` en el applet

http://librosysolucionarios.net

y deje que el estudiante intente la misma pregunta varias veces, hasta que esté correcta. Debe utilizarse un método separado para generar cada pregunta nueva. Este método debe llamarse una vez cuando el applet empiece a ejecutarse, y cada vez que el usuario responda correctamente a la pregunta. Todo el proceso de dibujar en el applet debe realizarse por el método `paint`.

6.32 El uso de las computadoras en la educación se conoce como *instrucción asistida por computadora (CAI)*. Un problema que se desarrolla en los entornos CAI es la fatiga de los estudiantes. Este problema puede eliminarse si se varía el diálogo de la computadora para mantener la atención del estudiante. Modifique el programa del ejercicio 6.31 de manera que los diversos comentarios se impriman para cada respuesta correcta e incorrecta, de la siguiente manera:

Contestaciones a una respuesta correcta:

```
¡Muy bien!
¡Excelente!
¡Buen trabajo!
¡Sigue así!
```

Contestaciones a una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez más.
¡No te rindas!
No. Sigue intentando.
```

Use la generación de números aleatorios para elegir un número entre 1 y 4 que se utilice para seleccionar una contestación apropiada a cada respuesta. Use una instrucción `switch` en el método `paint` para emitir las contestaciones.

6.33 Los sistemas de instrucción asistida por computadora más sofisticados supervisan el rendimiento del estudiante durante cierto tiempo. La decisión de empezar un nuevo tema se basa a menudo en el éxito del estudiante con los temas anteriores. Modifique el programa del ejercicio 6.32 para contar el número de respuestas correctas e incorrectas por parte del estudiante. Una vez que el estudiante escriba 10 respuestas, su programa debe calcular el porcentaje de respuestas correctas. Si éste es menor del 75%, imprima `Por favor pida ayuda adicional a su instructor` y reinicie el programa, para que otro estudiante pueda probarlo.

6.34 Escriba un applet que juegue a "adivina el número" de la siguiente manera: su programa elige el número a adivinar, seleccionando un entero aleatorio en el rango de 1 a 1000. El applet muestra el indicador `Adivine un número entre 1 y 1000` enseguida de un objeto `JTextField`. El jugador escribe su primer intento en el objeto `JTextField` y oprime *Intro*. Si la respuesta del jugador es incorrecta, su programa debe mostrar el mensaje `Demasiado alto. Intente de nuevo.` o `Demasiado bajo. Intente de nuevo.` en la barra de estado, para ayudar a que el jugador "se acerque" a la respuesta correcta. El programa debe borrar el objeto `JTextField` de manera que el usuario pueda escribir su siguiente intento. Cuando el usuario escriba la respuesta correcta, muestre el mensaje `¡Felicidades. Adivinó el número!` en la barra de estado y borre el objeto `JTextField` de manera que el usuario pueda jugar otra vez. (*Nota:* La técnica para adivinar empleada en este problema es similar a una *búsqueda binaria*.)

6.35 Modifique el programa del ejercicio 6.34 para contar el número de intentos que haga el jugador. Si el número es 10 o menos, imprima el mensaje `¡O ya sabía usted el secreto, o tuvo suerte!` Si el jugador adivina el número en 10 intentos, imprima el mensaje `¡Ajá! ¡Sabía usted el secreto!` Si el jugador hace más de 10 intentos, imprima el mensaje `¡Debería haberlo hecho mejor!` ¿Por qué no se deben requerir más de 10 intentos? Bueno, en cada "buen intento", el jugador debe poder eliminar la mitad de los números. Ahora muestre por qué cualquier número de 1 a 1000 puede adivinarse en 10 o menos intentos.

6.36 Escriba un método recursivo llamado `potencia( base, exponente )` que, al ser llamado, devuelva el resultado de

$$base^{exponente}$$

Por ejemplo, `potencia( 3, 4 ) = 3 * 3 * 3 * 3`. Suponga que `exponente` es un entero mayor o igual a 1. (*Sugerencia:* El paso de recursividad debe utilizar la relación

$$base^{exponente} = base \cdot base^{exponente - 1}$$

y la condición de terminación ocurre cuando `exponente` es igual a `1`, ya que

$$base^{1} = base$$

Incorpore este método en un applet que permita al usuario introducir la `base` y el `exponente`.)

6.37 (*Torres de Hanoi.*) Todo científico computacional en ciernes debe hacer frente a ciertos problemas clásicos, y el de las *Torres de Hanoi* (vea la figura 6.22) es uno de los más famosos. Cuenta la leyenda que, en un templo del Lejano Oriente, los sacerdotes tratan de mover una pila de discos de una estaca a otra. La estaca inicial tiene 64 discos ensartados y ordenados, de abajo hacia arriba, en orden de tamaño decreciente. Los sacerdotes intentan mover la pila de esta estaca hacia una segunda estaca, con las restricciones de mover sólo un disco a la vez, y que ningún disco más grande debe colocarse encima de uno más pequeño. Una tercera estaca está disponible para alojar discos temporalmente. En teoría el mundo se acabará cuando los sacerdotes completen su tarea, por lo que no hay muchos incentivos para que nosotros les facilitemos su trabajo.

Vamos a suponer que los sacerdotes intentan mover los discos de la estaca 1 a la 3. Queremos desarrollar un algoritmo que despliegue la secuencia precisa correspondiente a la transferencia de discos de una estaca a otra.

Si fuéramos a resolver este problema con los métodos convencionales, nos encontraríamos rápidamente a nosotros mismos involucrados en el manejo de los discos. En cambio, si atacamos el problema con la recursividad en mente, éste se vuelve inmediatamente manejable. El proceso de mover n discos puede considerarse en términos de mover solamente $n - 1$ discos (es decir, con recursividad) de la siguiente forma:

a) Mover $n - 1$ discos de la estaca 1 a la 2, utilizando la estaca 3 como un área de almacenamiento temporal.
b) Mover el último disco (el más grande) de la estaca 1 a la 3.
c) Mover los $n - 1$ discos de la estaca 2 a la 3, utilizando la estaca 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea involucra mover el disco $n = 1$ (es decir, el caso base). Esta tarea se logra moviendo simplemente el disco, sin necesidad de usar un área de almacenamiento temporal.

Escriba un applet para resolver el problema de las Torres de Hanoi. Debe permitir al usuario introducir el número de discos en un objeto `JTextField`. Use un método recursivo llamado `torre`, con cuatro parámetros:

a) El número de discos a mover.
b) La estaca en la que estarán ensartados inicialmente estos discos.
c) La estaca a la que se moverá esta pila de discos.
d) La estaca que se va a utilizar como área temporal de almacenamiento.

Su programa debe mostrar en un objeto `JTextArea`, con funcionalidad de desplazamiento, las instrucciones precisas que se necesitarán para mover los discos desde la estaca inicial hasta la estaca final. Por ejemplo, para mover una estaca de tres discos desde la estaca 1 hasta la estaca 3, su programa debe imprimir la siguiente serie de movimientos:

```
1 → 3 (Esta notación significa "mover un disco de la estaca 1 a la 3".)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3
```

6.38 Cualquier programa que pueda implementarse en forma recursiva, puede implementarse en forma iterativa, aunque algunas veces con mayor dificultad y menos claridad. Intente escribir una versión iterativa de las Torres de Hanoi.

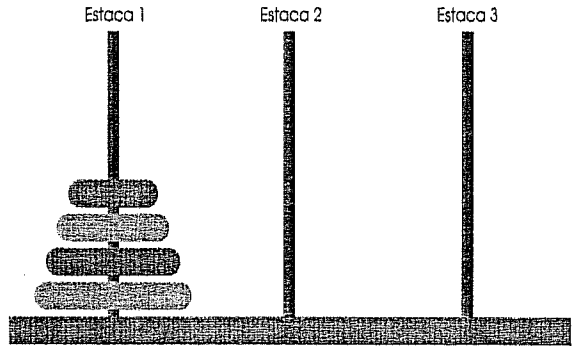


Figura 6.22 Las Torres de Hanoi para el caso con cuatro discos.

http://librosysolucionarios.net

Si tiene éxito, compare su versión iterativa con la versión recursiva que desarrolló en el ejercicio 6.37. Investigue cuestiones de rendimiento, claridad y su habilidad para demostrar que los programas sean correctos.

6.39 (*Visualización de la recursividad.*) Es interesante observar a la recursividad "en acción". Modifique el método factorial de la figura 6.15 para que imprima su variable local y el parámetro de la llamada recursiva. Por cada llamada recursiva, muestre los resultados en una línea separada y agregue un nivel de sangría. Haga su mejor esfuerzo para que los resultados sean claros, interesantes y útiles. Su objetivo aquí es diseñar e implementar un formato para los resultados que ayude a una persona a comprender mejor la recursividad. Tal vez quiera agregar dichas capacidades de despliegue a los muchos otros ejemplos y ejercicios de recursividad que contiene este libro.

6.40 El máximo común divisor de los enteros `x` y `y` es el entero más grande que puede dividir tanto a `x` como a `y`. Escriba un método recursivo llamado `mcd` que devuelva el máximo común divisor de `x` y `y`. El `mcd` de `x` y `y` se define recursivamente de la siguiente manera: si `y` es igual a `0`, entonces `mcd( x, y )` es `x`; de no ser así, `mcd( x, y )` es `mcd( y, x % y )`, en donde `%` es el operador residuo. Utilice este método para reemplazar al que escribió en el applet del ejercicio 6.28.

6.41 En los ejercicios 6.31 a 6.33 se desarrolló un programa de instrucción asistida por computadora para enseñar a un estudiante de escuela primara cómo multiplicar. Realice las siguientes mejoras:

a) Modifique el programa para que permita al usuario introducir un nivel de capacidad escolar. Un nivel de `1` significa que el programa debe usar sólo números de un dígito en los problemas, un nivel `2` significa que el programa debe utilizar números de dos dígitos máximo, etcétera.

b) Modifique el programa para permitir al usuario que elija el tipo de problemas aritméticos que desea estudiar. Una opción de `1` significa problemas de suma solamente, `2` significa problemas de resta, `3` significa problemas de multiplicación, `4` significa problemas de división y `5` significa una mezcla aleatoria de problemas de todos estos tipos.

6.42 Escriba un método llamado `distancia`, para calcular la distancia entre dos puntos $(x1, y1)$ y $(x2, y2)$. Todos los números y valores de retorno deben ser de tipo `double`. Incorpore este método en un applet que permita al usuario introducir las coordenadas de los puntos.

6.43 ¿Qué hace el siguiente método?

```
//  El parámetro b debe ser un entero
//  positivo para evitar la recursividad infinita
public int misterio( int a, int b )
{
   if ( b == 1 )
      return a;
   else
      return a + misterio( a, b - 1);
}
```

6.44 Una vez que determine lo que hace el programa del ejercicio 6.43, modifique el método para que opere apropiadamente, sin la restricción de que el segundo argumento debe ser no negativo. Además, incorpore el método en un applet que permita al usuario introducir dos enteros. Pruebe el método.

6.45 Encuentre el error en el siguiente método recursivo, y explique cómo corregirlo:

```
public int suma( int n )
{
   if ( n == 0 )
      return 0;
   else
      return n + suma( n );
}
```

6.46 Modifique el programa Craps de la figura 6.9 para permitir apuestas. Inicialice la variable `saldoBanco` con $1000. Pida al jugador que introduzca una `apuesta`. Compruebe que esa `apuesta` sea menor o igual al `saldoBanco` y, si no lo es, haga que el usuario vuelva a introducir la `apuesta` hasta que se introduzca un valor válido. Después de esto, comience un juego de craps. Si el jugador gana, agregue la `apuesta` al `saldoBanco` e imprima el nuevo `saldoBanco`. Si el jugador pierde, reste la `apuesta` al `saldoBanco`, imprima el nuevo

`saldoBanco`, compruebe si `saldoBanco` se ha vuelto cero y, de ser así, imprima el mensaje `"Lo siento. ¡Se quedó sin fondos!"` A medida que el juego progrese, imprima varios mensajes para crear algo de "charla", como `"Oh, se está yendo a la quiebra, ¿verdad?"`, o `"Oh, vamos, ¡arriésguese!"`, o `"La hizo en grande. ¡Ahora es tiempo de cambiar sus fichas por efectivo!"`. Implemente la "charla" como un método separado que seleccione en forma aleatoria la cadena a mostrar.

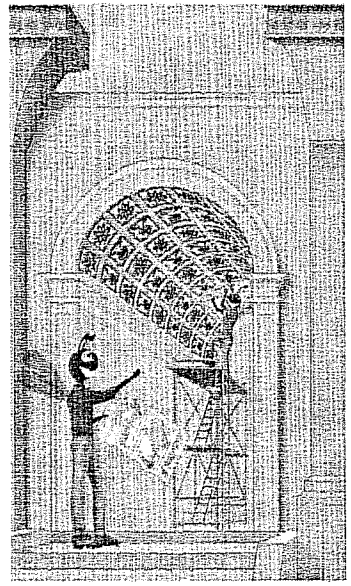
http://librosysolucionarios.net

7

Arreglos

Objetivos

- Introducir la estructura de datos conocida como arreglo.
- Comprender el uso de los arreglos para guardar, ordenar y buscar en listas y tablas de valores.
- Aprender a declarar un arreglo, inicializarlo y hacer referencia a sus elementos individuales.
- Pasar arreglos a los métodos.
- Declarar y manipular arreglos multidimensionales.



Con sollozos y lágrimas sorteó
Aquello de mayor tamaño...
Lewis Carroll

Intenta hasta el fin y nunca dudes;
Nada es tan difícil que no pueda averiguarse mediante la búsqueda.
Robert Herrick

Ahora ve, anótalo ante ellos en una tabla,
y escríbelo en un libro.
Isaías 30:8

Estará asegurado en mi memoria,
Y tú mismo(a) guardarás la llave.
William Shakespeare

http://librosysolucionarios.net

Plan general

7.1 Introducción
7.2 Arreglos
7.3 Declaración y creación de arreglos
7.4 Ejemplos acerca del uso de arreglos
7.5 Referencias y parámetros de referencia
7.6 Cómo pasar arreglos a los métodos
7.7 Ordenamiento de arreglos
7.8 Búsqueda en arreglos: Búsqueda lineal y búsqueda binaria
7.9 Arreglos multidimensionales
7.10 (Ejemplo práctico opcional) Acerca de los objetos: Colaboración entre los objetos

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Ejercicios de recursividad • Sección especial: Construya su propia computadora

7.1 Introducción

En este capítulo presentamos el importante tema de las *estructuras de datos* (colecciones de elementos de datos relacionados). Los *arreglos* son estructuras de datos que consisten de elementos de datos relacionados, del mismo tipo. Los arreglos son entidades de longitud fija; conservan siempre la misma longitud una vez creados, aunque la referencia a un arreglo puede reasignarse a un nuevo arreglo de distinta longitud. En el capítulo 20 veremos las estructuras dinámicas de datos como listas, colas, pilas y árboles, que pueden crecer y encoger a medida que se ejecutan los programas. En el capítulo 21 hablaremos sobre la clase `Vector`, que es una clase parecida a los arreglos y cuyos objetos pueden crecer y encoger en respuesta a un cambio en los requerimientos de almacenamiento de un programa en Java. En el capítulo 22 veremos las estructuras de datos predefinidas en Java que permiten al programador utilizar las estructuras de datos existentes para listas, colas, pilas y árboles. La API de colecciones también proporciona la clase `Arrays`, que cuenta con un conjunto de métodos utilitarios para manipular arreglos.

7.2 Arreglos

En Java, un arreglo es un grupo de variables (llamadas *elementos* o *componentes*) que contienen valores del mismo tipo. Recuerde que en la sección 3.5 vimos que los tipos en Java se dividen en dos categorías: *tipos primitivos* y *tipos de referencia*. Los arreglos en Java son objetos, por lo que se consideran como tipos de referencia. Los elementos de un arreglo en Java pueden ser tipos primitivos o de referencia (incluyendo arreglos, como veremos en la sección 7.9). Para hacer referencia a un elemento específico en un arreglo, debemos especificar el nombre de la referencia al arreglo y el número de la posición del elemento en el arreglo. El número de la posición del elemento se conoce formalmente como el *índice* o *subíndice* del elemento.

En la figura 7.1 se muestra una representación lógica de un arreglo de enteros, llamado `c`. Este arreglo contiene 12 elementos (es decir, variables). Un programa puede hacer referencia a cualquiera de estos elementos mediante una *expresión de acceso a un arreglo* que incluye el nombre del arreglo, seguido por el índice del elemento específico encerrado entre *corchetes* (`[]`). El primer elemento en cualquier arreglo tiene el *índice cero* (lo que se denomina como *elemento cero*). Por lo tanto, el primer elemento del arreglo `c` es `c[ 0 ]`, el segundo elemento es `c[ 1 ]`, el séptimo elemento del arreglo `c` es `c[ 6 ]` y, en general, el *i*-ésimo elemento del arreglo `c` es `c[ i - 1 ]`. Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables.

Un índice debe ser un entero positivo o una expresión entera que pueda promoverse a un `int`. Si un programa utiliza una expresión como índice, el programa evalúa la expresión para determinar el índice. Por ejemplo, si suponemos que la variable `a` es `5` y que `b` es `6`, entonces la instrucción:

```
c[ a + b ] += 2;
```

http://librosysolucionarios.net

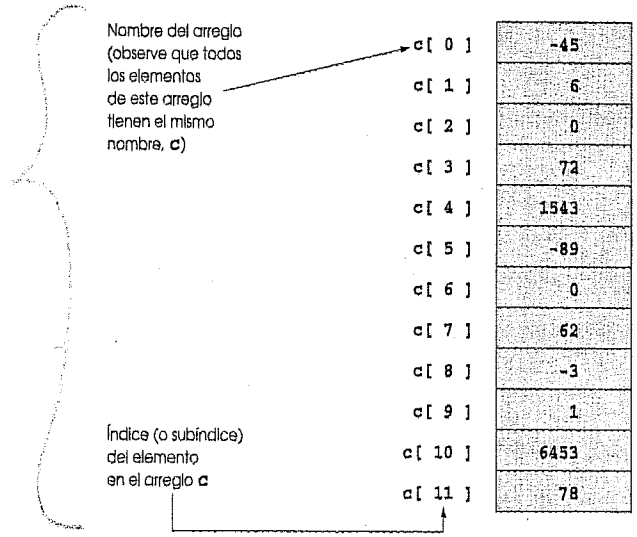


Figura 7.1 Un arreglo de 12 elementos.

suma 2 al elemento `c[ 11 ]` del arreglo. Observe que el nombre del arreglo con subíndice es una expresión de acceso al arreglo. Dichas expresiones pueden utilizarse en el lado izquierdo de una asignación, para colocar un nuevo valor en un elemento del arreglo.



Error común de programación 7.1

Usar un valor de tipo `long` como índice de un arreglo produce un error de compilación.

Examinemos de manera más detallada el arreglo `c` de la figura 7.1. El *nombre* de la referencia al arreglo es `c`. Todo objeto arreglo en Java *conoce* su propia longitud y mantiene esta información en un campo del objeto arreglo, llamado `length`. La expresión `c.length` accede al campo `length` del arreglo `c` para determinar la longitud del arreglo. La manera en que se hace referencia a los 12 elementos de este arreglo es: `c[ 0 ]`, `c[ 1 ]`, `c[ 2 ]`, ..., `c[ 11 ]`. El *valor* de `c[ 0 ]` es `-45`, el valor de `c[ 1 ]` es `6`, el valor de `c[ 2 ]` es `0`, el valor de `c[ 7 ]` es `62` y el valor de `c[ 11 ]` es `78`. Para calcular la suma de los valores contenidos en los primeros tres elementos del arreglo `c` y guardar el resultado en la variable `suma`, escribiríamos:

```
suma = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Para dividir el valor del séptimo elemento del arreglo `c` entre 2 y asignar el resultado a la variable `x`, escribiríamos:

```
x = c[ 6 ] / 2;
```



Error común de programación 7.2

Observe la diferencia entre el "séptimo elemento del arreglo" y el "elemento siete del arreglo". Los índices de los arreglos empiezan en 0, por lo que el "séptimo elemento del arreglo" tiene un índice de 6, mientras que el "elemento siete del arreglo" tiene un índice de 7 y es en realidad el octavo elemento del arreglo. Esta confusión es fuente de errores de "desplazamiento en uno".

7.3 Declaración y creación de arreglos

Los objetos arreglo ocupan espacio en memoria. Todos los objetos en Java (incluyendo los arreglos) deben crearse con la palabra clave `new` (como se describió en la sección 4.9). El programador especifica el tipo de

cada elemento y el número de elementos que se requieren para el arreglo, como parte de una *expresión para crear un arreglo* que utiliza la palabra clave `new`. La siguiente declaración y expresión crea 12 elementos para el arreglo de enteros `c` de la figura 7.1:

```
int c[] = new int[ 12 ];
```

Esta tarea también puede realizarse en dos pasos, como se muestra a continuación:

```
int c[];           // declara la variable arreglo
c = new int[ 12 ]; // crea el arreglo
```

Al crear un arreglo, cada uno de sus elementos recibe un valor predeterminado: cero para los elementos numéricos de tipos primitivos, `false` para los elementos `boolean` y `null` para las referencias (cualquier tipo no primitivo).

Error común de programación 7.3



A diferencia de las declaraciones de arreglos en algunos otros lenguajes de programación (como C y C++), las declaraciones de arreglos en Java no especifican el número de elementos del arreglo en los corchetes que van después del nombre del arreglo (por ejemplo, `int c[ 12 ];`); en caso contrario, se produciría un error de sintaxis.

Un programa puede crear varios arreglos en una sola declaración. La siguiente declaración de un arreglo `String` reserva 100 elementos para `b` y 27 para `x`:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

Al declarar un arreglo, su tipo y los corchetes pueden combinarse al principio de la declaración para indicar que todos los identificadores en la declaración son referencias a arreglos. Por ejemplo,

```
double[] arreglo1, arreglo2;
```

declara a `arreglo1` y `arreglo2` como referencias a arreglos de valores `double`. Como vimos anteriormente, la declaración y creación del arreglo pueden combinarse en la declaración. La siguiente declaración reserva 10 elementos para el `arreglo1` y 20 para el `arreglo2`:

```
double[] arreglo1 = new double[ 10 ], arreglo2 = new double[ 20 ];
```

Un programa puede declarar arreglos de cualquier tipo. Todo elemento de un arreglo de tipo primitivo es una variable del tipo declarado del arreglo. Por ejemplo, cada uno de los elementos de un arreglo `int` es una variable `int`. En un arreglo que sea de un tipo de referencia, cada elemento del arreglo es una referencia a un objeto del tipo declarado de ese arreglo. Por ejemplo, cada uno de los elementos de un arreglo `String` es una referencia a un objeto `String`.

7.4 Ejemplos acerca del uso de arreglos

En esta sección presentaremos varios ejemplos que muestran cómo declarar, crear e inicializar arreglos y cómo manipular sus elementos. Por cuestión de simpleza, los ejemplos en esta sección utilizan arreglos que contienen elementos de tipo `int`. Tenga en cuenta que los programas pueden crear arreglos de cualquier tipo.

Cómo crear e inicializar un arreglo

En la aplicación de la figura 7.2 se utiliza la palabra clave `new` para crear un arreglo de 10 elementos `int`, los cuales inicialmente tienen el valor cero (el valor predeterminado para las variables `int`). El programa muestra los elementos del arreglo con un formato tabular, en un objeto de la clase `JTextArea`.

```
1  // Fig. 7.2: InicArreglo.java
2  // Creación de un arreglo.
3  import javax.swing.*;
4
5  public class InicArreglo {
6
7     public static void main( String args[] )
8     {
9        int arreglo[]; // declarar la referencia a un arreglo
10
11       arreglo = new int[ 10 ]; // crear el arreglo
12
13       String salida = "Índice\tValor\n";
14
15       // anexar el valor de cada elemento del arreglo al String salida
16       for ( int contador = 0; contador < arreglo.length; contador++ )
17          salida += contador + "\t" + arreglo[ contador ] + "\n";
18
19       JTextArea areaSalida = new JTextArea();
20       areaSalida.setText( salida );
21
22       JOptionPane.showMessageDialog( null, areaSalida,
23          "Inicialización de un arreglo de valores int",
24          JOptionPane.INFORMATION_MESSAGE );
25
26       System.exit( 0 );
27
28    } // fin de main
29
30 } // fin de la clase InicArreglo
```

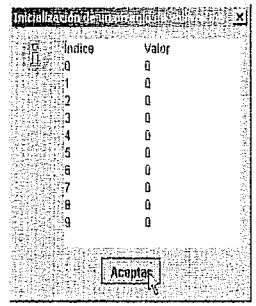


Figura 7.2 Inicialización de los elementos de un arreglo en ceros. (Parte 1 de 2.)

Figura 7.2 Inicialización de los elementos de un arreglo en ceros. (Parte 2 de 2.)

En la línea 9 se declara `arreglo`, una referencia capaz de referirse a un arreglo de elementos `int`. En la línea 11 se crea el arreglo y se le asigna la referencia `arreglo` al objeto arreglo resultante. El programa crea su salida en el objeto `String` llamado `salida` que se mostrará en un objeto `JTextArea` de un cuadro de diálogo de mensaje. En la línea 13 se asignan a `salida` los encabezados de las columnas que mostrará el programa. La primera columna representa el índice para cada elemento del arreglo, y la segunda el valor de cada elemento del arreglo.

En las líneas 16 y 17 se utiliza una instrucción `for` para anexar a `salida` el número de índice (representado por `contador`) y el valor de cada elemento del arreglo (representado por `arreglo[ contador ]`). Observe que el ciclo utiliza un contador basado en cero (recuerde, los valores de los índices empiezan en 0), de manera que el ciclo pueda utilizar todos los elementos del arreglo. Además, en la condición del `for` se utiliza la expresión `arreglo.length` para determinar la longitud del arreglo. En este ejemplo la longitud del arreglo es 10, por lo que el ciclo continúa ejecutándose siempre y cuando el valor de la variable de control `contador`

http://librosysolucionarios.net

sea menor que 10. Para un arreglo de 10 elementos los valores de los índices son del 0 al 9, por lo que al utilizar el operador menor se garantiza que el ciclo no intente acceder a un elemento más allá del final del arreglo.

Uso de un inicializador de arreglo

Como alternativa a la expresión para crear un arreglo, un programa puede crear un arreglo e inicializar sus elementos con un *inicializador de arreglo*, que es una lista de expresiones separadas por comas (la cual se conoce también como *lista inicializadora*) encerrada entre llaves ({ y }). En este caso, la longitud del arreglo se determina en base al número de elementos en la lista inicializadora. Por ejemplo, la declaración:

```
int n[] = { 10, 20, 30, 40, 50 };
```

crea un arreglo de cinco elementos con los valores de índices 0, 1, 2, 3 y 4. Esta declaración no requiere de new para crear el objeto arreglo. Cuando el compilador encuentra la declaración de un arreglo que incluye una lista inicializadora, el compilador cuenta el número de inicializadores en la lista para determinar el número apropiado de elementos del arreglo.

La aplicación de la figura 7.3 inicializa un arreglo de enteros con 10 valores (línea 11) y muestra el arreglo en formato tabular en un objeto `JTextArea` de un cuadro de diálogo de mensaje. El código para mostrar los elementos del arreglo (líneas 13 a 24) es idéntico al de la figura 7.2.

```
1  // Fig. 7.3: InicArreglo.java
2  //Inicialización de un arreglo con una declaración.
3  import javax.swing.*;
4
5  public class InicArreglo {
6
7     public static void main( String args[] )
8     {
9        // la lista inicializadora especifica el número de elementos y
10       // el valor para cada elemento
11       int arreglo[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
12
13       String salida = "Índice\tValor\n";
14
15       // anexar cada valor de los elementos del arreglo al String salida
16       for ( int contador = 0; contador < arreglo.length; contador++ )
17          salida += contador + "\t" + arreglo[ contador ] + "\n";
18
19       JTextArea areaSalida = new JTextArea();
20       areaSalida.setText( salida );
21
22       JOptionPane.showMessageDialog( null, areaSalida,
23          "Inicialización de un arreglo con una declaración",
24          JOptionPane.INFORMATION_MESSAGE );
25
26       System.exit( 0 );
27
28    } // fin de main
29
30 } // fin de la clase InicArreglo
```

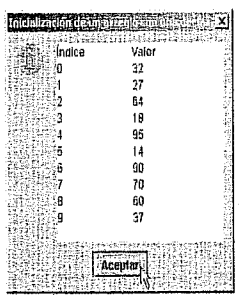


Figura 7.3 Inicialización de los elementos de un arreglo con una declaración.

Cálculo del valor a guardar en cada elemento del arreglo

Algunos programas calculan el valor almacenado en cada uno de los elementos de un arreglo. La aplicación de la figura 7.4 crea un arreglo de 10 elementos y asigna a cada uno de ellos uno de los enteros pares del 2 al 20 (2, 4, 6, ..., 20). Después, el programa muestra el arreglo en formato tabular. La instrucción for de las líneas 15 a 16 genera el valor de un elemento del arreglo, multiplicando por 2 el valor actual de la variable de control del ciclo for, llamada contador, y sumándole 2.

En la línea 9 se utiliza el calificador `final` para declarar la constante `LONGITUD_ARREGLO`, cuyo valor es `10`. Recuerde que en la sección 6.8 vimos que las constantes deben inicializarse antes de poder utilizarse, y que no pueden modificarse a partir de entonces. Si se intenta modificar una variable `final` una vez que se haya declarado, como se muestra en la línea 11, el compilador generará un mensaje como:

```
cannot assign a value to final variable nombreVariable
```

```
1  // Fig. 7.4: InicArreglo.java
2  // Inicialización de un arreglo con los enteros pares del 2 al 20.
3  import javax.swing.*;
4
5  public class InicArreglo {
6
7     public static void main( String args[] )
8     {
9        final int LONGITUD_ARREGLO = 10;    // constante
10       int arreglo[];                      // referencia a un arreglo int
11
12       arreglo = new int[ LONGITUD_ARREGLO ]; // crear el arreglo
13
14       // calcular el valor para cada elemento del arreglo
15       for ( int contador = 0; contador < arreglo.length; contador++ )
16          arreglo[ contador ] = 2 + 2 * contador;
17
18       String salida = "Índice\tValor\n";
19
20       for ( int contador = 0; contador < arreglo.length; contador++ )
21          salida += contador + "\t" + arreglo[ contador ] + "\n";
22
23       JTextArea areaSalida = new JTextArea();
24       areaSalida.setText( salida );
25
26       JOptionPane.showMessageDialog( null, areaSalida,
27          "Inicialización con enteros pares del 2 al 20",
28          JOptionPane.INFORMATION_MESSAGE );
29
30       System.exit( 0 );
31
32    } // fin de main
33
34 } // fin de la clase InicArreglo
```

Figura 7.4 Generación de valores para colocarlos en los elementos de un arreglo. (Parte 1 de 2.)

http://librosysolucionarios.net

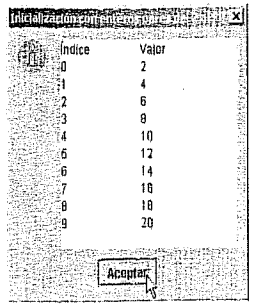


Figura 7.4 Generación de valores para colocarlos en los elementos de un arreglo. (Parte 2 de 2.)

Si intenta utilizar una variable `final` antes de inicializarla, el compilador genera el siguiente mensaje de error:

```
Variable nombreVariable may not have been initialized
```

Las constantes también se conocen como *constantes con nombre* o *variables de sólo lectura*. Este tipo de variables a menudo facilitan la legibilidad de los programas.



Error común de programación 7.4

Asignar un valor a una constante después de que se ha inicializado producirá un error de compilación.

Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores que se emplearán en un cálculo. Por ejemplo, si los elementos del arreglo representan las calificaciones de un examen, tal vez el profesor desee sumar el total de los elementos del arreglo y utilizar esa suma para calcular el promedio del examen.

La aplicación de la figura 7.5 suma los valores contenidos en el arreglo de 10 elementos de tipo entero. El programa declara, crea e inicializa el arreglo en la línea 9. La instrucción `for` en las líneas 13 y 14 realiza los cálculos. (*Nota:* Los valores suministrados como inicializadores de arreglos generalmente se introducen en un programa, en vez de especificarse en una lista inicializadora. Por ejemplo, el usuario de un subprograma podría introducir los valores a través de un objeto `JTextField`, o en una aplicación los valores podrían leerse de un archivo en el disco [como veremos en el capítulo 17]. Al hacer que los datos se introduzcan como entrada en el programa éste se hace más flexible, ya que puede utilizarse con distintos conjuntos de datos.)

```
1  // Fig. 7.5: SumarArreglo.java
2  // Sumar el total de los valores de los elementos de un arreglo.
3  import javax.swing.*;
4
5  public class SumarArreglo {
6
7     public static void main( String args[] )
8     {
9        int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10       int total = 0;
11
12       // sumar el valor de cada elemento al total
13       for ( int contador = 0; contador < arreglo.length; contador++ )
14          total += arreglo[ contador ];
15
```

Figura 7.5 Cálculo de la suma de los elementos de un arreglo. (Parte 1 de 2.)

http://librosysolucionarios.net


```
16       JOptionPane.showMessageDialog( null,
17          "Total de elementos del arreglo: " + total,
18          "Sumar los elementos de un arreglo",
19          JOptionPane.INFORMATION_MESSAGE );
20
21       System.exit( 0 );
22
23    } // fin de main
24
25 } // fin de la clase SumarArreglo
```

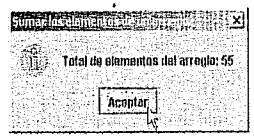


Figura 7.5 Cálculo de la suma de los elementos de un arreglo. (Parte 2 de 2.)

Uso de histogramas para mostrar los datos de un arreglo en forma gráfica

Muchos programas presentan datos a los usuarios en forma gráfica. Por ejemplo, a menudo los valores numéricos se muestran como barras en un gráfico de barras, o *histograma*. En este tipo de gráficos, las barras más largas representan los valores numéricos más grandes. Una manera simple de mostrar datos numéricos en forma gráfica es con un histograma que muestre cada valor numérico como una barra de asteriscos (`*`).

Nuestra siguiente aplicación (figura 7.6) lee números de un arreglo y grafica la información en forma de un histograma. El programa muestra cada valor seguido por una barra que consiste de varios asteriscos que corresponden al valor numérico. Las instrucciones `for` anidadas (líneas 14 a 21) anexan las barras a la cadena `salida`, la cual se muestra en el área de texto `areaSalida` de un cuadro de diálogo de mensaje.

Observe la condición de la instrucción `for` en la línea 18 (`estrellas < arreglo[ contador ]`). Cada vez que el programa llega a esa instrucción `for` interna (líneas 18 y 19), el ciclo cuenta desde 0 hasta `arreglo[ contador ]`, utilizando así un valor en `arreglo` para determinar el valor final de la variable de control `estrellas` y el número de asteriscos a mostrar.

```
1  // Fig. 7.6: Histograma.java
2  // Programa para imprimir histogramas.
3  import javax.swing.*;
4
5  public class Histograma {
6
7     public static void main( String args[] )
8     {
9        int arreglo[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
10
11       String salida = "Elemento\tValor\tHistograma";
12
13       // para cada elemento del arreglo, mostrar una barra en el histograma
14       for ( int contador = 0; contador < arreglo.length; contador++ ) {
15          salida += "\n" + contador + "\t" + arreglo[ contador ] + "\t";
16
17          // imprimir barra de asteriscos
18          for ( int estrellas = 0; estrellas < arreglo[ contador ]; estrellas++ )
19             salida += "*";
```

Figura 7.6 Un programa que imprime histogramas. (Parte 1 de 2.)

```
20
21        } // fin de instrucción for externa
22
23        JTextArea areaSalida = new JTextArea();
24        areaSalida.setText( salida );
25
26        JOptionPane.showMessageDialog( null, areaSalida,
27           "Programa para imprimir histogramas", JOptionPane.INFORMATION_MESSAGE );
28
29        System.exit( 0 );
30
31     } // fin de main
32
33  } // fin de la clase Histograma
```

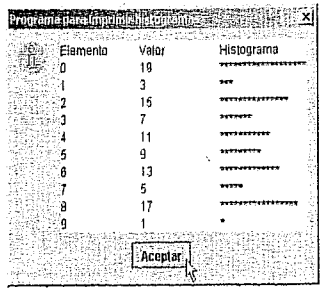


Figura 7.6 Un programa que imprime histogramas. (Parte 2 de 2.)

Uso de los elementos de un arreglo como contadores

En ocasiones, los programas utilizan una serie de variables contadores para sintetizar datos como los resultados de una encuesta. En la figura 6.8 utilizamos una serie de contadores en nuestro programa para tirar dados, para rastrear el número de ocurrencias de cada lado en un dado de seis lados, a medida que el programa tiraba el dado 6000 veces. También indicamos que hay una técnica más elegante para sintetizar los valores de los dados. En la figura 7.7 se muestra una versión de la aplicación de la figura 6.8, esta vez utilizando arreglos.

```
1  // Fig. 7.7: TirarDado.java
2  // Tirar un dado de seis lados 6000 veces.
3  import javax.swing.*;
4
5  public class TirarDado {
6
7     public static void main( String args[] )
8     {
9        int frecuencia[] = new int[ 7 ];
10
11       // tirar un dado 6000 veces; usar el valor del dado como índice de frecuencia
12       for ( int tiro = 1; tiro <= 6000; tiro++ )
13          ++frecuencia[ 1 + ( int ) ( Math.random() * 6 ) ];
14
15       String salida = "Cara\tFrecuencia";
16
```

Figura 7.7 Programa para tirar dados utilizando arreglos en vez de `switch`. (Parte 1 de 2.)

http://librosysolucionarios.net


```
17       // anexar frecuencias al String salida
18       for ( int cara = 1; cara < frecuencia.length; cara++ )
19          salida += "\n" + cara + "\t" + frecuencia[ cara ];
20
21       JTextArea areaSalida = new JTextArea();
22       areaSalida.setText( salida );
23
24       JOptionPane.showMessageDialog( null, areaSalida,
25          "Tirar un dado 6000 veces", JOptionPane.INFORMATION_MESSAGE );
26
27       System.exit( 0 );
28
29    } // fin de main
30
31 } // fin de la clase TirarDado
```

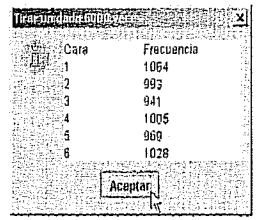


Figura 7.7 Programa para tirar dados utilizando arreglos en vez de `switch`. (Parte 2 de 2.)

El programa utiliza el arreglo de siete elementos llamado `frecuencia` (línea 9) para contar las ocurrencias de cada lado del dado. La línea 13 de este programa reemplaza a las líneas 17 a 43 de la figura 6.8. En la línea 13 se utiliza el valor aleatorio como índice del arreglo `frecuencia`, para determinar qué elemento debe incrementar el programa durante cada iteración del ciclo. El cálculo del número aleatorio en la línea 13 produce números del 1 al 6 (los valores para un dado de seis lados), por lo que el arreglo `frecuencia` debe ser lo suficientemente grande como para almacenar seis contadores. Sin embargo, en este programa optamos por utilizar un arreglo de siete elementos. Ignoramos el primer elemento del arreglo, `frecuencia[ 0 ]`, ya que es más lógico hacer que el valor 1 para la cara incremente a `frecuencia[ 1 ]`, en vez de `frecuencia[ 0 ]`. Esto nos permite utilizar el valor para cada cara directamente como un índice para el arreglo `frecuencia`.



Buena práctica de programación 7.1

Siempre procure que sus programas sean claros. En ocasiones, es necesario sacrificar el uso más eficiente de la memoria o tiempo del procesador para lograr que sus programas sean más claros.

Por otro lado, las líneas 18 y 19 de este programa reemplazan a las líneas 49 a 52 de la figura 6.8. Gracias a que podemos iterar a través del arreglo `frecuencia`, no fue necesario enumerar, como lo hicimos en la figura 6.8, cada línea de texto que mostramos en el objeto `JTextArea`.

Uso de arreglos para analizar los resultados de una encuesta

En nuestro siguiente ejemplo utilizaremos arreglos para sintetizar los resultados de la información recolectada en una encuesta. Considere el siguiente problema:

Se pidió a cuarenta estudiantes que calificaran la calidad de la comida en la cafetería, en una escala del 1 al 10 (en donde 1 significa pésimo y 10 significa excelente). Coloque las 40 respuestas en un arreglo entero y sintetice los resultados de la encuesta.

La solución a este problema es una típica aplicación de procesamiento de arreglos (vea la figura 7.8). Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 10). El arreglo **respuestas** es de tipo entero de 40 elementos, y contiene las respuestas de los estudiantes a la encuesta. Utilizamos un arreglo de 11 elementos, llamado **frecuencia**, para contar el número de ocurrencias de cada respuesta. Al igual que en la figura 7.7, ignoramos el primer elemento (**frecuencia[0]**) ya que es más lógico hacer que la respuesta 1 incremente a **frecuencia[1]** en vez de **frecuencia[0]**. Esto nos permite utilizar cada respuesta directamente como subíndice del arreglo **frecuencia**. Cada elemento del arreglo se utiliza como un contador para una de las respuestas de la encuesta.

El ciclo **for** de las líneas 16 y 17 toma las respuestas, una a la vez, del arreglo **respuestas** e incrementa uno de los 10 contadores en el arreglo **frecuencia** (desde **frecuencia[1]** hasta **frecuencia [10]**). La instrucción clave en el ciclo es la línea 17, en la cual se incrementa el contador apropiado de **frecuencia**, dependiendo del valor de **respuestas[respuesta]**.

Consideremos varias iteraciones del ciclo **for**. Cuando la variable de control **respuesta** es 0, el valor de **respuestas[respuesta]** es el valor del primer elemento del arreglo **respuestas** (es decir, 1), por lo que el programa interpreta la instrucción **++frecuencia[respuestas[respuesta]];** como:

```
++frecuencia[ 1 ];
```

```
1  // Fig. 7.8: VotacionEstudiantes.java
2  // Programa de votación de los estudiantes.
3  import javax.swing.*;
4
5  public class VotacionEstudiantes {
6
7     public static void main( String args[] )
8     {
9        int respuestas[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
10          10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
11          4, 8, 6, 8, 10 };
12       int frecuencia[] = new int[ 11 ];
13
14       // para cada respuesta, seleccionar el elemento de respuestas y usar ese valor
15       // como índice en frecuencia para determinar el elemento a incrementar
16       for ( int respuesta = 0; respuesta < respuestas.length; respuesta++ )
17          ++frecuencia[ respuestas[ respuesta ] ];
18
19       String salida = "Calificación\tFrecuencia\n";
20
21       // anexar frecuencias al String salida
22       for ( int calificacion = 1; calificacion < frecuencia.length; calificacion++ )
23          salida += calificacion + "\t" + frecuencia[ calificacion ] + "\n";
24
25       JTextArea areaSalida = new JTextArea();
26       areaSalida.setText( salida );
27
28       JOptionPane.showMessageDialog( null, areaSalida,
29          "Programa de votación de estudiantes", JOptionPane.INFORMATION_MESSAGE );
30
31       System.exit( 0 );
32
33    } // fin de main
34
35 } // fin de la clase VotacionEstudiantes
```

Figura 7.8 Un programa sencillo de análisis de votos estudiantiles. (Parte 1 de 2.)

http://librosysolucionarios.net

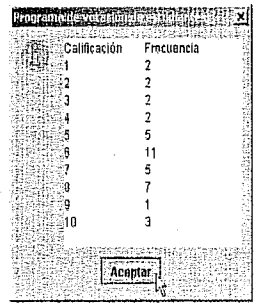


Figura 7.8 Un programa sencillo de análisis de votos estudiantiles. (Parte 2 de 2.)

con lo cual se incrementa el valor en el elemento uno del arreglo. Para evaluar la expresión, empiece con el valor en el conjunto más interior de corchetes (**respuesta**). Una vez que sepa el valor de **respuesta**, insértelo en la expresión y evalúe el siguiente conjunto exterior de corchetes (es decir, **respuestas[respuesta]**. Después utilice el valor resultante como índice en el arreglo **frecuencia**, para determinar cuál contador incrementar.

Cuando **respuesta** es 1, **respuestas[respuesta]** es el valor del segundo elemento (2) de **respuestas**, por lo que el programa interpreta la instrucción **++frecuencia[respuestas[respuesta]];** como:

```
++frecuencia[ 2 ];
```

con lo cual se incrementa el elemento dos del arreglo (el tercer elemento del arreglo).

Cuando **respuesta** es 2, **respuestas[respuesta]** es el valor del tercer elemento (6) de **respuestas**, por lo que el programa interpreta la instrucción **++frecuencia[respuestas[respuesta]];** como:

```
++frecuencia[ 6 ];
```

con lo cual se incrementa el elemento seis del arreglo (el séptimo elemento del arreglo), y así sucesivamente. Sin importar el número de respuestas procesadas en la encuesta, el programa requiere sólo un arreglo de 11 elementos (ignorando al elemento cero) para sintetizar los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 10, y los valores del índice de un arreglo de 11 elementos son del 0 al 10. Además, observe que los resultados sintetizados son correctos, ya que los elementos del arreglo **frecuencia** se inicializaron con cero cuando se creó el arreglo con **new**.

Si los datos tuvieran valores inválidos como 13, el programa habría intentado sumar 1 a **frecuencia [13]**. Este elemento está fuera de los límites del arreglo. En los lenguajes de programación C y C++, una referencia así hubiera sido permitida por el compilador y en tiempo de ejecución. El programa "caminaría" más allá del final del arreglo, en donde se hubiera encontrado el elemento 13, en caso de haber existido, y sumaría 1 al valor que se encontrara en esa ubicación en memoria. Esta operación podría llegar a modificar otra variable en el programa, o incluso podría ocasionar que el programa terminara prematuramente. Java impide el acceso a los elementos fuera de los límites de un arreglo.



Tip para prevenir errores 7.1

Cuando se ejecuta un programa en Java, el intérprete de Java comprueba los índices de los arreglos para asegurar que sean válidos (es decir, todos los índices de arreglos deben ser mayores o iguales a 0, y menores que la longitud del arreglo). Si hay un índice inválido, Java genera una excepción.



Tip para prevenir errores 7.2

Las excepciones indican que ocurrió un error en un programa. Un programador puede escribir código para recuperarse de una excepción y continuar con la ejecución del programa, en vez de que éste termine en forma anormal. Cuando un programa intenta acceder a un elemento fuera de los límites de un arreglo, Java genera una excepción **ArrayIndexOutOfBoundsException**.[1]

1. Hablaremos sobre el manejo de excepciones en el capítulo 15.

Tip para prevenir errores 7.3

Al iterar a través de un arreglo, el índice del arreglo nunca debe pasar por debajo de 0 y siempre debe ser menor que la longitud del arreglo. La condición de terminación de ciclo debe evitar el acceso a los elementos fuera de este rango.

7.5 Referencias y parámetros de referencia

En esta sección demostraremos cómo pasar arreglos y elementos de arreglos como argumentos para los métodos. Primero presentaremos los mecanismos utilizados para pasar argumentos a los métodos. Las dos formas de pasar argumentos a los métodos en muchos lenguajes de programación (como C y C++) son el *paso por valor* y el *paso por referencia* (también conocidos como *llamada por valor* y *llamada por referencia*). Cuando se pasa un argumento por valor, se hace una *copia* del argumento y ésta se pasa al método llamado.

Tip para prevenir errores 7.4

Con el paso por valor, los cambios realizados en la copia del método llamado no afectan al valor original de la variable en el método que hizo la llamada. Esto evita los efectos secundarios accidentales que dificultan enormemente el desarrollo de sistemas de software correctos y confiables.

Cuando se pasa un argumento por referencia, el método solicitante da al método llamado la habilidad de acceder a los datos del solicitante directamente, por lo que existe la posibilidad de que modifique esos datos. El paso por referencia también mejora el rendimiento, al eliminar la sobrecarga de tener que copiar grandes cantidades de datos.

Observación de ingeniería de software 7.1

A diferencia de otros lenguajes, Java no permite al programador elegir entre pasar cada argumento por valor o por referencia. Las variables de tipo primitivo siempre se pasan por valor. Los objetos no se pasan a los métodos, sino que se pasan referencias a esos objetos. Las referencias se pasan por valor; esto es, al método se le pasa la copia de una referencia. Con una referencia a un objeto, el método puede manipular directamente a ese objeto.

Observación de ingeniería de software 7.2

Al devolver información de un método mediante una instrucción `return`*, los tipos primitivos siempre se devuelven por valor (es decir, se devuelve una copia) los objetos siempre se devuelven por referencia (es decir, se devuelve una referencia al objeto).*

Para pasar a un método la referencia de un objeto, simplemente especifique el nombre de la referencia en la llamada al método. El nombre del parámetro del método correspondiente en realidad hace referencia al objeto en memoria, por lo cual el objeto original puede utilizarse directamente en el método invocado.

Recuerde que en la sección 7.2 vimos que, en Java, los arreglos son objetos; por lo tanto, los arreglos se pasan a los métodos por referencia (un método que sea llamado puede utilizar los elementos de los arreglos originales del método que hizo la llamada). El nombre de una variable arreglo es en realidad una referencia a un objeto que contiene los elementos del arreglo y el campo `length`, una constante que indica el número de elementos en el arreglo. En la siguiente sección mostraremos cómo se utiliza el paso de arreglos por valor y por referencia.

Tip de rendimiento 7.1

Pasar arreglos por referencia tiene sentido por cuestiones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Para los arreglos grandes de uso frecuente, esto desperdiciaría tiempo y consumiría una considerable cantidad de almacenamiento para las copias de los arreglos.

7.6 Cómo pasar arreglos a los métodos

Para pasar un arreglo como argumento a un método, se especifica el nombre del arreglo sin corchetes. Por ejemplo, si el arreglo `temperaturasPorHora` se declara como:

```
int temperaturasPorHora[] = new int[ 24 ];
```


entonces, la llamada al método

```
modificarArreglo( temperaturasPorHora );
```

pasa al método `modificarArreglo`, una referencia al arreglo `temperaturasPorHora`. En Java, todos los objetos arreglo "conocen" su propia longitud (mediante el campo `length`). Por lo tanto, cuando pasamos un objeto arreglo a un método, no tenemos que pasar la longitud del arreglo como un argumento adicional.

Aunque los arreglos completos (y los objetos a los que se hace referencia mediante elementos individuales de arreglos tipo referencias) se pasan por referencia, los elementos individuales de los arreglos de tipos primitivos se pasan por valor, de la misma forma que se pasan las variables simples. Dichos valores primitivos se conocen como *escalares* o *cantidades escalares*. Para pasar el elemento de un arreglo a un método, utilice el nombre del arreglo y el subíndice entre corchetes como argumento en la llamada al método.

Para que un método reciba un arreglo mediante una llamada, la lista de parámetros del método debe especificar un parámetro tipo arreglo (o varios, si se va a recibir más de un arreglo). Por ejemplo, el encabezado del método `modificarArreglo` podría escribirse así:

```
void modificarArreglo( int b[] )
```

indicando que `modificarArreglo` espera recibir un arreglo entero en el parámetro `b`. Como los arreglos se pasan por referencia, cuando el método llamado utiliza el nombre del arreglo `b`, se está refiriendo al arreglo en sí (`temperaturasPorHora` en la llamada anterior) en el método que está haciendo la llamada.

El applet de la figura 7.9 muestra la diferencia entre pasar un arreglo completo y pasar el elemento de un arreglo de tipo primitivo. Una vez más, utilizamos un applet aquí porque todavía no hemos definido una aplicación que contenga otros métodos aparte de `main`. Aún estamos aprovechando algunas características de los applets, como la creación automática de un objeto applet y las llamadas a `init`, `start` y `paint` realizadas por el contenedor de applets. En el capítulo 9 introduciremos aplicaciones que se ejecutan en sus propias ventanas. Hasta entonces empezaremos a ver clases de aplicaciones que contienen varios métodos.

```
1  // Fig. 7.9: PasoArreglos.java
2  // Paso de arreglos y elementos individuales de un arreglo a métodos.
3  import java.awt.Container;
4  import javax.swing.*;
5
6  public class PasoArreglos extends JApplet {
7
8     // inicializar applet
9     public void init()
10    {
11       JTextArea areaSalida = new JTextArea();
12       Container contenedor = getContentPane();
13       contenedor.add( areaSalida );
14
15       int arreglo[] = { 1, 2, 3, 4, 5 };
16
17       String salida = "Efectos de pasar el arreglo completo por referencia:\n" +
18          "Los valores del arreglo original son:\n";
19
20       // anexar los elementos del arreglo original al String salida
21       for ( int contador = 0; contador < arreglo.length; contador++ )
22          salida += "   " + arreglo[ contador ];
23
24       modificarArreglo( arreglo );  // arreglo pasado por referencia
25
26       salida += "\n\nLos valores del arreglo modificado son:\n";
```

Figura 7.9 Paso de arreglos y elementos individuales de un arreglo a métodos. (Parte 1 de 2.)

http://librosysolucionarios.net

```
27
28          // anexar los elementos del arreglo modificado al String salida
29          for ( int contador = 0; contador < arreglo.length; contador++ )
30             salida += "   " + arreglo[ contador ];
31
32          salida += "\n\nEfectos de pasar el elemento de un arreglo por valor:\n" +
33             "arreglo[3] antes de modificarElemento: " + arreglo[ 3 ];
34
35          modificarElemento( arreglo[ 3 ] );  // intento de modificar arreglo[ 3 ]
36
37          salida += "\narreglo[3] después de modificarElemento: " + arreglo[ 3 ];
38          areaSalida.setText( salida );
39
40       } // fin del método init
41
42       // multiplicar cada elemento de un arreglo por 2
43       public void modificarArreglo( int arreglo2[] )
44       {
45          for ( int contador = 0; contador < arreglo2.length; contador++ )
46             arreglo2[ contador ] *= 2;
47       }
48
49       // multiplicar argumento por 2
50       public void modificarElemento( int elemento )
51       {
52          elemento *= 2;
53       }
54
55    } // fin de la clase PasoArreglos
```

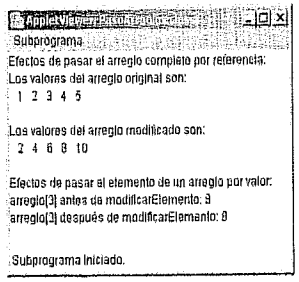


Figura 7.9 Paso de arreglos y elementos individuales de un arreglo a métodos. (Parte 2 de 2.)

En las líneas 11 a 13 del método `init` se crea el objeto `JTextArea` llamado `areaSalida`, y se adjunta al panel de contenido del applet. La instrucción `for` en las líneas 21 y 22 anexa los cinco elementos de `arreglo` (un arreglo de valores `int`) al objeto `String` llamado `salida`. En la línea 24 se invoca el método `modificarArreglo`, pasándole `arreglo` como argumento. El método `modificarArreglo` (líneas 43 a 47) multiplica cada uno de los elementos por dos. Para mostrar que los elementos de `arreglo` se modificaron, la instrucción `for` en las líneas 29 y 30 anexa los cinco elementos de `arreglo` a `salida` nuevamente. Como se muestra en la figura, el método `modificarArreglo` cambió el valor de cada elemento.

Después el programa muestra que los elementos individuales de los arreglos de tipo primitivo se pasan a los métodos por valor. Para mostrar el valor de `arreglo[ 3 ]` antes de llamar al método `modificarElemento`, en las líneas 32 y 33 se anexa el valor de `arreglo[ 3 ]` (el cuarto elemento con el valor 8) a `salida`. En la línea 35 se invoca el método `modificarElemento` y se le pasa `arreglo[ 3 ]` como argumento. Recuerde que `arreglo[ 3 ]` es en realidad un valor `int` (8) en `arreglo`. Además, recuerde que los valores de los tipos primitivos se pasan a los métodos por valor. Por lo tanto, el programa pasa una copia de `arreglo[ 3 ]`. El método `modificarElemento` multiplica su argumento por dos y guarda el resultado (16) en su parámetro `elemento`. Los parámetros de los métodos, al igual que las variables locales, dejan de existir cuando el método en el que se declaran termina su ejecución. Por lo tanto, cuando el método `modificarElemento` termina, el parámetro `elemento` del método se destruye. De esta forma, cuando el programa devuelve el control a `init`, en la línea 37 se anexa el valor, sin modificar, de `arreglo[ 3 ]` (es decir, 8) a `salida`. En la línea 38 se muestran los resultados en el objeto `JTextArea`.

7.7 Ordenamiento de arreglos

El *ordenamiento* de los datos (es decir, colocar los datos en un orden específico, como ascendente o descendente) es una de las aplicaciones computacionales más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que pueda preparar estados de cuenta individuales al final de cada mes. Las compañías telefónicas ordenan sus listas de cuentas por apellido y, además, por nombre para facilitar el proceso de búsqueda de números telefónicos. Virtualmente todas las organizaciones necesitan ordenar datos y, en muchos casos, cantidades masivas de ellos. El ordenamiento de datos es un problema intrigante que atrae algunos de los esfuerzos de investigación más intensos en el campo de la ciencia computacional. En este capítulo hablaremos sobre uno de los esquemas de ordenamiento más sencillos. En los ejercicios de éste y de los capítulos 20 y 22, investigaremos esquemas más complejos que producen un rendimiento superior.

En la figura 7.10 se acomodan los valores de `arreglo` (un arreglo con 10 elementos de variables `int`) en orden ascendente. La técnica que utilizamos se conoce como *ordenamiento burbuja* o *método de hundimiento*, ya que los valores más pequeños "flotan" gradualmente hacia arriba, hasta llegar a la parte superior del arreglo (es decir, hacia el primer elemento) como las burbujas de aire que se elevan hacia la superficie del agua, mientras que los valores más grandes se hunden en el fondo (final) del arreglo. Esta técnica utiliza ciclos anidados para recorrer el arreglo varias veces. En cada pasada se comparan pares sucesivos de elementos. Si un par se encuentra en orden ascendente (o los valores son iguales), el ordenamiento burbuja deja los valores como están. Si un par se encuentra en orden descendente, el ordenamiento burbuja intercambia sus valores en el arreglo. El applet contiene los métodos `init`, `ordenamBurbuja` e `intercambiar`. El método `init` (líneas 9 a 33) inicializa el applet. El método `ordenamBurbuja` (líneas 36 a 55) se llama desde `init` para ordenar los elementos de `arreglo`. El método `ordenamBurbuja` llama al método `intercambiar` (líneas 58 a 65) según sea necesario para intercambiar dos elementos del arreglo.

```
1  // Fig. 7.10: OrdenamBurbuja.java
2  // Ordenamiento de los valores de un arreglo en orden ascendente.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class OrdenamBurbuja extends JApplet {
7
8     // inicializar applet
9     public void init()
10    {
11       JTextArea areaSalida = new JTextArea();
12       Container contenedor = getContentPane();
13       contenedor.add( areaSalida );
14
15       int arreglo[] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16
17       String salida = "Elementos de datos en su orden original\n";
18
```

Figura 7.10 Ordenamiento de un arreglo mediante el método burbuja. (Parte 1 de 2.)

http://librosysolucionarios.net

```
19          // anexar los valores originales al String salida
20          for ( int contador = 0; contador < arreglo.length; contador++ )
21             salida += "   " + arreglo[ contador ];
22
23          ordenamBurbuja( arreglo ); // ordenar arreglo
24
25          salida += "\n\nElementos de datos en orden ascendente\n";
26
27          // anexar los valores ordenados del arreglo al String salida
28          for ( int contador = 0; contador < arreglo.length; contador++ )
29             salida += "   " + arreglo[ contador ];
30
31          areaSalida.setText( salida );
32
33       } // fin del método init
34
35       // ordenar elementos del arreglo con el método burbuja
36       public void ordenamBurbuja( int arreglo2[] )
37       {
38          // ciclo para controlar número de pasadas
39          for ( int pasada = 1; pasada < arreglo2.length; pasada++ ) {
40
41             // ciclo para controlar número de comparaciones
42             for ( int elemento = 0;
43                   elemento < arreglo2.length - 1;
44                   elemento++ ) {
45
46                // comparar elementos uno a uno e intercambiarlos si
47                // el primer elemento es mayor que el segundo
48                if ( arreglo2[ elemento ] > arreglo2[ elemento + 1 ] )
49                   intercambiar( arreglo2, elemento, elemento + 1 );
50
51             } // fin del ciclo para controlar las comparaciones
52
53          } // fin del ciclo para controlar las pasadas
54
55       } // fin del método ordenamBurbuja
56
57       // intercambiar dos elementos de un arreglo
58       public void intercambiar( int arreglo3[], int primero, int segundo )
59       {
60          int almacen; // área temporal de almacenamiento para intercambiar
61
62          almacen = arreglo3[ primero ];
63          arreglo3[ primero ] = arreglo3[ segundo ];
64          arreglo3[ segundo ] = almacen;
65       }
66
67    } // fin de la clase OrdenamBurbuja
```

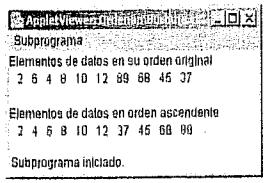


Figura 7.10 Ordenamiento de un arreglo mediante el método burbuja. (Parte 2 de 2.)

http://librosysolucionarios.net

En las líneas 20 y 21 se anexan los valores originales de `arreglo` al objeto `String` llamado `salida`. En la línea 23 se invoca el método `ordenamBurbuja` y se le pasa `arreglo` como el arreglo a ordenar. El método recibe el arreglo como el parámetro `arreglo2`. La instrucción `for` anidada de las líneas 39 a 53 realiza el ordenamiento. El ciclo externo controla el número de pasadas del arreglo. El ciclo interno controla las comparaciones y el intercambio (si es necesario) de los elementos durante cada pasada. El método `ordenamBurbuja` compara `arreglo2[ 0 ]` con `arreglo2[ 1 ]`, luego `arreglo2[ 1 ]` con `arreglo2[ 2 ]`, luego `arreglo2[ 2 ]` con `arreglo2[ 3 ]` y así, sucesivamente, hasta que completa la pasada comparando `arreglo2[ 8 ]` con `arreglo2[ 9 ]`. Aunque hay 10 elementos, el ciclo de comparación realiza solamente nueve comparaciones. Durante cada pasada, un valor grande podría desplazarse hacia abajo por el arreglo (hundirse) muchas posiciones. Sin embargo, un valor pequeño puede avanzar (flotar) sólo una posición hacia arriba en cada pasada. En la primera pasada se garantiza que el valor más grande se hundirá hasta el elemento en el fondo del arreglo, `arreglo2[ 9 ]`. En la segunda pasada, el segundo valor más grande se hundirá hasta `arreglo2[ 8 ]`. En la novena pasada, el noveno valor más grande se hundirá hasta `arreglo2[ 1 ]`, dejando el valor más pequeño en `arreglo2[ 0 ]`. Por lo tanto, sólo se requieren nueve pasadas para ordenar un arreglo de 10 elementos.

Si una comparación revela que los dos elementos están en orden descendente, en la línea 49 el método `ordenamBurbuja` llama al método `intercambiar` para intercambiar los dos elementos, de manera que se encuentren en orden ascendente en el arreglo. El método `intercambiar` recibe una referencia al arreglo (a la cual llama `arreglo3`) y dos enteros que representan los índices de los dos elementos del arreglo que van a intercambiarse. El intercambio se lleva a cabo mediante las tres asignaciones de las líneas 62 a 64. La variable adicional `almacen` guarda temporalmente uno de los dos valores que van a intercambiarse. El intercambio no puede realizarse sólo con las dos asignaciones

```
arreglo3[ primero ] = arreglo3 [ segundo ];
arreglo3[ segundo ] = arreglo3 [ primero ];
```

Si `arreglo3[ primero ]` es 7 y `arreglo3[ segundo ]` es 5, después de la primera asignación ambos elementos del arreglo contienen 5 y el valor 7 se pierde; por ende, se necesita la variable adicional `almacen`.

La virtud principal del método burbuja es que es fácil de programar. Sin embargo, este método se ejecuta lentamente; esto se vuelve aparente cuando se ordenan arreglos extensos. En el ejercicio 7.11 le pediremos que desarrolle versiones más eficientes del método burbuja. Otros ejercicios analizan algunos algoritmos de ordenamiento que son mucho más eficientes que el método burbuja.[2]

Tip de rendimiento 7.2

En ocasiones, los algoritmos más simples tienen un rendimiento pobre; sin embargo, su virtud es que son fáciles de programar, probar y depurar. Pero algunas veces se requieren algoritmos más complejos para llegar a un máximo rendimiento.

7.8 Búsqueda en arreglos: Búsqueda lineal y búsqueda binaria

A menudo los programadores trabajan con grandes cantidades de datos almacenadas en arreglos. Podría ser necesario determinar si un arreglo contiene un valor que coincide con cierto *valor clave*. El proceso de localizar un valor clave en un arreglo se denomina *búsqueda*. En esta sección veremos dos técnicas de búsqueda: la *búsqueda lineal* simple y la *búsqueda binaria*, que es más eficiente.[3]

Cómo buscar en un arreglo mediante la búsqueda lineal

En el applet de la figura 7.11, el método `busquedaLineal` (declarado en las líneas 47 a 58) utiliza una instrucción `for` (líneas 50 a 54) que contiene una instrucción `if` para iterar sobre los elementos de un arreglo, y comparar cada elemento con una *clave de búsqueda*: el valor a localizar en el arreglo. Si se encuentra la clave

2. En los cursos más avanzados (a menudo titulados "Estructuras de datos"; "Algoritmos" o "Complejidad computacional") se analizan el ordenamiento y la búsqueda con más detalle.

3. En los ejercicios 7.33 y 7.34 le pedimos que implemente versiones recursivas de la búsqueda lineal y la búsqueda binaria, respectivamente.

de búsqueda el método devuelve el índice del elemento, indicando por ende la posición exacta de la clave de búsqueda en el arreglo. Si la clave de búsqueda no se encuentra, el método devuelve el valor `-1`. Regresamos `-1` ya que no es un índice válido. Si el arreglo en el que se realiza la búsqueda no se encuentra en ningún orden específico, existe la misma probabilidad de que la clave de búsqueda se encuentre tanto en el primer elemento que como en el último. Por lo tanto, en promedio el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del arreglo.

La figura 7.11 contiene un arreglo de 100 elementos relleno con los enteros pares del 0 al 198. El usuario escribe la clave de búsqueda en un objeto **`JTextField`** y oprime *Intro (Entrar)* para empezar la búsqueda. Pasamos una referencia al arreglo como argumento para el método **`busquedaLineal`**, a pesar de que el arreglo puede utilizarse directamente en ese método, mediante el campo **`arreglo`** de la clase **`BusquedaLineal`**. Hacemos esto porque una referencia a un arreglo generalmente se pasa a un método de otra clase para buscar el arreglo correspondiente. Por ejemplo, la clase **`Arrays`** (vea el capítulo 22) contiene una variedad de métodos **`static`** para ordenar arreglos; buscar en arreglos; comparar el contenido de arreglos; y llenar arreglos de tipos primitivos, arreglos de tipo **`Object`** y arreglos de tipo **`String`**.

```
1  // Fig. 7.11: BusquedaLineal.java
2  // Búsqueda lineal en un arreglo.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class BusquedaLineal extends JApplet implements ActionListener {
8
9     JLabel escribirEtiqueta, resultadoEtiqueta;
10    JTextField escribirCampo, resultadoCampo;
11    int arreglo[];
12
13    // configurar la GUI del applet
14    public void init()
15    {
16       // obtener panel de contenido y configurar su esquema como FlowLayout
17       Container contenedor = getContentPane();
18       contenedor.setLayout( new FlowLayout() );
19
20       // configurar JLabel y JTextField para la entrada del usuario
21       escribirEtiqueta = new JLabel( "Escriba clave de búsqueda (entero)" );
22       contenedor.add( escribirEtiqueta );
23
24       escribirCampo = new JTextField( 10 );
25       contenedor.add( escribirCampo );
26
27       // registrar este applet para escuchar acciones de escribirCampo
28       escribirCampo.addActionListener( this );
29
30       // configurar JLabel y JTextField para mostrar los resultados
31       resultadoEtiqueta = new JLabel( "Resultado" );
32       contenedor.add( resultadoEtiqueta );
33
34       resultadoCampo = new JTextField( 20 );
35       resultadoCampo.setEditable( false );
36       contenedor.add( resultadoCampo );
37
38       // crear arreglo y llenarlo con enteros pares de 0 a 198
```

Figura 7.11 Búsqueda lineal en un arreglo. (Parte 1 de 2.)

http://librosysolucionarios.net


```
39       arreglo = new int[ 100 ];
40
41       for ( int contador = 0; contador < arreglo.length; contador++ )
42          arreglo[ contador ] = 2 * contador;
43
44    } // fin del método init
45
46    // buscar en arreglo el valor clave especificado
47    public int busquedaLineal( int arreglo2[], int clave )
48    {
49       // iterar a través de los elementos del arreglo
50       for ( int contador = 0; contador < arreglo2.length; contador++ )
51
52          // si el elemento del arreglo es igual al valor de la clave,
              devolver ubicación
53          if ( arreglo2[ contador ] == clave )
54             return contador;
55
56       return -1;  // clave no encontrada
57
58    } // fin del método busquedaLineal
59
60    // obtener entrada del usuario y llamar al método busquedaLineal
61    public void actionPerformed( ActionEvent actionEvent )
62    {
63       // la entrada puede obtenerse también mediante escribirCampo.getText()
64       String claveBusqueda = actionEvent.getActionCommand();
65
66       // pasar referencia de arreglo a busquedaLineal; generalmente se pasa
            la referencia
67       // a un arreglo a un método, para buscar el objeto arreglo correspondiente
68       int elemento = busquedaLineal( arreglo, Integer.parseInt( claveBusqueda ) );
69
70       // mostrar el resultado de la búsqueda
71       if ( elemento != -1 )
72          resultadoCampo.setText( "Valor encontrado en el elemento " + elemento );
73       else
74          resultadoCampo.setText( "Valor no encontrado" );
75
76    } // fin del método actionPerformed
77
78 } // fin de la clase BusquedaLineal
```

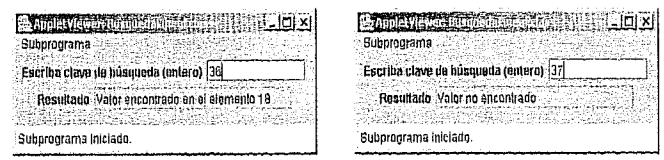


Figura 7.11 Búsqueda lineal en un arreglo. (Parte 2 de 2.)

Cómo buscar en un arreglo ordenado mediante la búsqueda binaria

El método de búsqueda lineal funciona bien con arreglos pequeños o desordenados. Sin embargo, para arreglos extensos la búsqueda lineal es ineficiente. Si el arreglo está ordenado, podemos utilizar la técnica de búsqueda binaria, de alta velocidad.

Como vimos en el ejemplo anterior, el algoritmo de búsqueda lineal compara la clave de búsqueda con un promedio de la mitad de los elementos del arreglo. Sin embargo, el algoritmo de búsqueda binaria elimina la mitad de los elementos que se van a buscar en el arreglo, después de cada comparación. El algoritmo localiza el elemento medio del arreglo y lo compara con la clave de búsqueda. Si son iguales, la clave de búsqueda se ha encontrado y la búsqueda binaria devuelve el índice de ese elemento. En caso contrario, la búsqueda binaria reduce el problema a buscar en la mitad del arreglo ordenado. Si la clave de búsqueda es menor que el elemento medio del arreglo, se buscará en la primera mitad del arreglo; en caso contrario, se buscará en la segunda mitad.[4] Si la clave de búsqueda no es el elemento medio del subarreglo especificado (es decir, parte del arreglo original), el algoritmo se repite sobre una cuarta parte del arreglo original. La búsqueda continúa hasta que la clave de búsqueda sea igual al elemento medio de un subarreglo, o hasta que el subarreglo consista en un elemento que no sea igual a la clave de búsqueda (es decir, que no se encuentre la clave de búsqueda).

En el peor de los casos, para buscar un arreglo ordenado de 1023 elementos se requerirán solamente 10 comparaciones si se utiliza una búsqueda binaria. Al dividir varias veces 1024 entre 2 (ya que después de cada comparación, podemos eliminar la mitad del arreglo) se producen los valores 512, 256, 128, 64, 32, 16, 8, 4, 2 y 1. El número 1024 (2^{10}) se divide entre 2 sólo 10 veces para obtener el valor de 1. La división entre 2 es equivalente a una comparación en el algoritmo de búsqueda binaria. Por lo tanto, un arreglo de 1,048,576 (2^{20}) elementos requiere un máximo de 20 comparaciones para encontrar la clave, y un arreglo de mil millones de elementos requiere un máximo de 30 comparaciones para encontrar la clave. Éste es un tremendo aumento en rendimiento, en comparación con la búsqueda lineal. Para un arreglo de mil millones de elementos, ¡ésta es una diferencia entre un promedio de 500 millones de comparaciones y un máximo de 30 comparaciones! El número máximo de comparaciones necesarias para la búsqueda binaria de cualquier arreglo ordenado es el exponente de la primera potencia de 2 que sea mayor que el número de elementos en el arreglo.

En la figura 7.12 presentamos un método **busquedaBinaria** iterativo (líneas 78 a 109), el cual recibe como argumentos un arreglo entero llamado **arreglo2** (el arreglo para la búsqueda) y una **clave** entera (la clave de búsqueda). El programa pasa, al método llamado **busquedaBinaria**, una referencia al arreglo. Una vez más, hacemos esto porque una referencia a un arreglo generalmente se pasa a un método de otra clase para buscar en el arreglo correspondiente. En **busquedaBinaria**, si **clave** concuerda con el elemento **medio** de un subarreglo, **busquedaBinaria** devuelve **medio** (el índice del elemento actual) para indicar que el valor se ha encontrado y que la búsqueda está completa. Si **clave** no concuerda con el elemento **medio** de un subarreglo, **busquedaBinaria** ajusta el índice **inferior** o **superior** (ambos declarados en el método) para continuar la búsqueda, utilizando un subarreglo más pequeño. Si **clave** es menor que el elemento medio, el índice **superior** se establece en **medio - 1**, y la búsqueda continúa con los elementos desde **inferior** hasta **medio - 1**. Si **clave** es mayor que el elemento medio, el índice **inferior** se establece en **medio + 1** y la búsqueda continúa con los elementos desde **medio + 1** hasta **superior**. La instrucción **if...else** anidada en las líneas 93 a 102 realiza estas comparaciones.

```
1  // Fig. 7.12: BusquedaBinaria.java
2  // Búsqueda binaria en un arreglo.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.text.*;
6
7  import javax.swing.*;
8
9  public class BusquedaBinaria extends JApplet implements ActionListener {
10    JLabel escribirEtiqueta, resultadoEtiqueta;
11    JTextField escribirCampo, resultadoCampo;
```

Figura 7.12 Búsqueda binaria en un arreglo ordenado. (Parte 1 de 4.)

4. Este ejemplo asume que el arreglo está en orden ascendente.

http://librosysolucionarios.net


```
12    JTextArea salida;
13
14    int arreglo[];
15    String pantalla = "";
16
17    // configurar la GUI del applet
18    public void init()
19    {
20       // obtener panel de contenido y configurar su esquema como FlowLayout
21       Container contenedor = getContentPane();
22       contenedor.setLayout( new FlowLayout() );
23
24       // configurar JLabel y JTextField para la entrada del usuario
25       escribirEtiqueta = new JLabel( "Escriba clave de búsqueda (entero)" );
26       contenedor.add( escribirEtiqueta );
27
28       escribirCampo = new JTextField( 10 );
29       contenedor.add( escribirCampo );
30
31       // registrar este applet para escuchar acciones de escribirCampo
32       escribirCampo.addActionListener( this );
33
34       // configurar JLabel y JTextField para mostrar los resultados
35       resultadoEtiqueta = new JLabel( "Resultado" );
36       contenedor.add( resultadoEtiqueta );
37
38       resultadoCampo = new JTextField( 20 );
39       resultadoCampo.setEditable( false );
40       contenedor.add( resultadoCampo );
41
42       // configurar JTextArea para mostrar datos de comparación
43       salida = new JTextArea( 6, 60 );
44       salida.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
45       contenedor.add( salida );
46
47       // crear arreglo y llenarlo con enteros pares de 0 a 28
48       arreglo = new int[ 15 ];
49
50       for ( int contador = 0; contador < arreglo.length; contador++ )
51          arreglo[ contador ] = 2 * contador;
52
53    } // fin del método init
54
55    // obtener entrada del usuario y llamar al método busquedaBinaria
56    public void actionPerformed( ActionEvent actionEvent )
57    {
58       // la entrada también puede obtenerse mediante escribirCampo.getText()
59       String claveBusqueda = actionEvent.getActionCommand();
60
61       // inicializar cadena pantalla para la nueva búsqueda
62       pantalla = "Porciones del arreglo buscadas\n";
63
64       // realizar búsqueda binaria
65       int elemento = busquedaBinaria( arreglo, Integer.parseInt( claveBusqueda ) );
66
```

Figura 7.12 Búsqueda binaria en un arreglo ordenado. (Parte 2 de 4.)

```
67          salida.setText( pantalla );
68
69          // mostrar resultado de la búsqueda
70          if ( elemento != -1 )
71             resultadoCampo.setText( "Valor encontrado en el elemento " + elemento );
72          else
73             resultadoCampo.setText( "Valor no encontrado" );
74
75       } // fin del método actionPerformed
76
77       // método para realizar la búsqueda binaria en un arreglo
78       public int busquedaBinaria( int arreglo2[], int clave )
79       {
80          int inferior = 0;                    // índice del elemento inferior
81          int superior = arreglo2.length - 1;  // índice del elemento superior
82          int medio;                           // índice del elemento medio
83
84          // iterar hasta que el índice inferior sea mayor que el superior
85          while ( inferior <= superior ) {
86             medio = ( inferior + superior ) / 2;  // determinar índice medio
87
88             // mostrar subconjunto de elementos del arreglo utilizados en esta
89             // iteración del ciclo de búsqueda binaria
90             crearSalida( arreglo2, inferior, medio, superior );
91
92             // si clave concuerda con elemento medio, devolver su ubicación
93             if ( clave == arreglo[ medio ] )
94                return medio;
95
96             // si clave es menor que elemento medio, establecer nuevo elemento superior
97             else if ( clave < arreglo[ medio ] )
98                superior = medio - 1;
99
100            // clave mayor que elemento medio, establecer nuevo elemento inferior
101            else
102               inferior = medio + 1;
103
104         } // fin de instrucción while
105
106         return -1;   // clave no encontrada
107
108      } // fin del método busquedaBinaria
109
110      // crear fila de salida mostrando subconjunto de elementos del arreglo
111      // que se están procesando
112      void crearSalida( int arreglo3[], int inferior, int medio, int superior )
113      {
114         // crear formato numérico entero de 2 dígitos
115         DecimalFormat twoDigits = new DecimalFormat( "00" );
116
117         // iterar a través de los elementos del arreglo
118         for ( int contador = 0; contador < arreglo3.length; contador++ ) {
119
120            // si contador está fuera de subconjunto actual del arreglo, anexar
121            // espacios de relleno al String pantalla
```

Figura 7.12 Búsqueda binaria en un arreglo ordenado. (Parte 3 de 4.)


```
122            if ( contador < inferior || contador > superior )
123               pantalla += "     ";
124
125            // si elemento medio, anexar elemento al String pantalla
126            // seguido de un asterisco (*), para indicar el elemento medio
127            else if ( contador == medio )
128               pantalla += twoDigits.format( arreglo3[ contador ] ) + "* ";
129
130            else // anexar elemento al String pantalla
131               pantalla += twoDigits.format( arreglo3[ contador ] ) + "  ";
132
133         } // fin de instrucción for
134
135         pantalla += "\n";
136
137      } // fin del método crearSalida
138
139   } // fin de la clase BusquedaBinaria
```

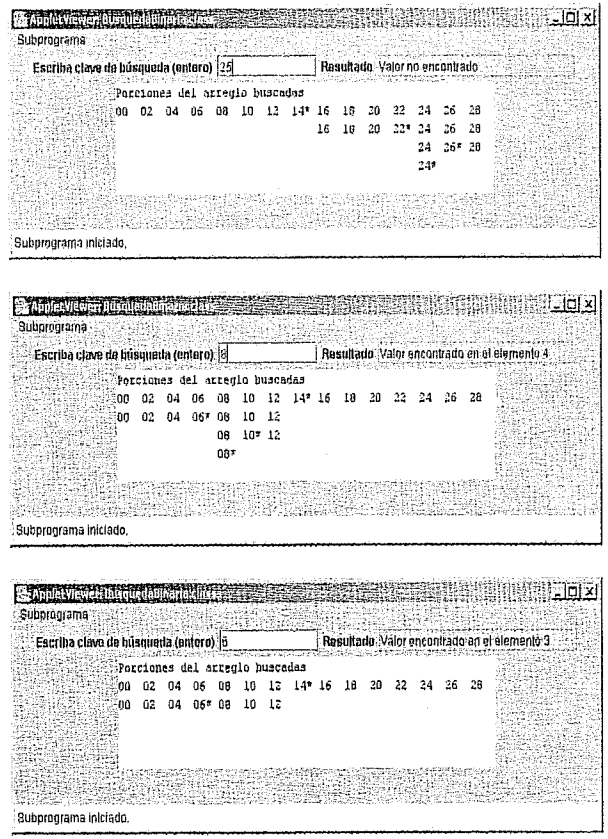


Figura 7.12 Búsqueda binaria en un arreglo ordenado. (Parte 4 de 4.)

http://librosysolucionarios.net

Este programa utiliza un arreglo de 15 elementos. La primera potencia de 2 mayor que el número de elementos es 16 (2^4), por lo que `busquedaBinaria` requiere cuando mucho cuatro comparaciones para encontrar la `clave`. Para ilustrar esto, en la línea 90 se realiza una llamada al método `crearSalida` (declarado en las líneas 112 a 137) para mostrar cada subarreglo durante el proceso de la búsqueda binaria. El método `crearSalida` marca el elemento medio en cada subarreglo con un asterisco (`*`) para indicar el elemento con el que se compara la `clave`. Sin importar cuál clave de búsqueda se introduzca, cada búsqueda en este ejemplo produce un máximo de cuatro líneas de salida: uno por cada comparación.

El área de texto `salida` utiliza un tipo de letra *Monoespaciado* (un *tipo de letra con anchura fija*; es decir, todos los caracteres son de la misma anchura) para ayudar a alinear el texto que aparece en cada línea de salida. En la línea 44 se utiliza el método `setFont` para cambiar el tipo de letra mostrado en `salida`. El método `setFont` puede cambiar el tipo de letra del texto que se muestra en la mayoría de los componentes de la GUI. Este método requiere un objeto `Font` (paquete `java.awt`) como argumento; dicho objeto se inicializa con tres argumentos: el nombre `String` del tipo de letra (`"Monospaced"`), un `int` que representa el estilo del tipo de letra (`Font.PLAIN` es una constante entera declarada en la clase `Font`, la cual indica un tipo de letra simple) y un `int` que representa el tamaño de punto del tipo de letra (`12`). Java cuenta con nombres genéricos para varias fuentes disponibles en todas las plataformas Java. El tipo de letra *monoespaciado* también se conoce como *Courier*. Otros tipos de letra comunes incluyen *Serif* (también conocido como *TimesRoman*) y *SansSerif* (también conocido como *Helvetica*). Java le ofrece acceso a todos los tipos de letra en su sistema mediante los métodos de la clase `GraphicsEnvironment` (paquete `java.awt`). El estilo también puede ser `Font.BOLD`, `Font.ITALIC` o `Font.BOLD + Font.ITALIC`. El tamaño de punto representa el tamaño del tipo de letra. Hay 72 puntos en una pulgada. El tamaño actual del texto al aparecer en la pantalla puede variar, con base en el tamaño de la pantalla y su resolución.

7.9 Arreglos multidimensionales

Los arreglos multidimensionales con dos dimensiones se utilizan a menudo para representar *tablas* de valores, que constan de información ordenada en *filas* y *columnas*. Para identificar un elemento específico de una tabla, debemos especificar dos índices. Por convención, el primer índice identifica la fila del elemento y el segundo, su columna. Los arreglos que requieren dos índices para identificar un elemento específico se llaman *arreglos bidimensionales*. (Los arreglos multidimensionales pueden tener más de dos dimensiones.) Java no soporta directamente el uso de arreglos multidimensionales, pero sí permite al programador especificar arreglos unidimensionales cuyos elementos sean también arreglos unidimensionales, logrando así el mismo efecto. En la figura 7.13 se muestra un arreglo bidimensional llamado `a`, el cual contiene tres filas y cuatro columnas (es decir, un arreglo de tres por cuatro). En general, un arreglo con *m* filas y *n* columnas se conoce como un *arreglo de m por n*.

Cada elemento del arreglo `a` se identifica en la figura 7.13 mediante una expresión de acceso a un arreglo, de la forma `a[fila][columna]`; `a` es el nombre de la referencia al arreglo, `fila` y `columna` son los índices que identifican en forma única a cada elemento en el arreglo `a` mediante un número de fila y columna.

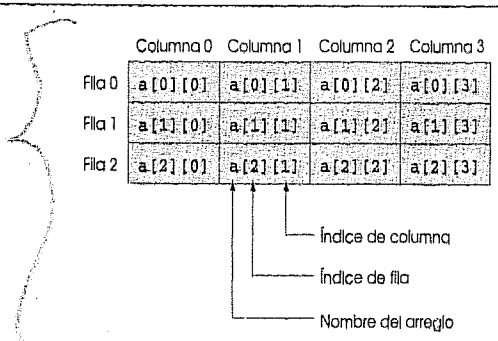


Figura 7.13 Arreglo bidimensional con tres filas y cuatro columnas.

http://librosysolucionarios.net

Observe que los nombres de los elementos en la primera fila tienen todos un primer índice de `0`; los nombres de los elementos en la cuarta columna tienen todos un segundo índice de `3`.

Arreglos de arreglos unidimensionales

Al igual que los arreglos unidimensionales, los arreglos multidimensionales pueden inicializarse mediante inicializadores de arreglos en las declaraciones. Un arreglo bidimensional llamado `b` con dos filas y dos columnas, podría declararse e inicializarse con *inicializadores de arreglos anidados*, como se muestra a continuación:

```
int b[][] = { { 1, 2 }, { 3, 4 } };
```

Los valores inicializadores se agrupan por fila entre corchetes. Por lo tanto, `1` y `2` inicializan `b[0][0]` y `b[0][1]`; `3` y `4` inicializan `b[1][0]` y `b[1][1]`. El compilador determina el número de filas contando el número de inicializadores de arreglos anidados (representados por conjuntos de llaves dentro de las llaves externas) en el inicializador del arreglo. El compilador determina el número de columnas en una fila contando los valores inicializadores en el inicializador del arreglo para esa fila.

En Java, los arreglos multidimensionales se mantienen como arreglos de arreglos unidimensionales. Por lo tanto, el arreglo `b` en la declaración anterior en realidad está compuesto de tres arreglos unidimensionales separados. El arreglo `b` en sí es un arreglo unidimensional que contiene dos elementos. Cada elemento es una referencia a un arreglo unidimensional de variables `int`.

Arreglos bidimensionales con filas de distintas longitudes

La forma en que Java representa los arreglos multidimensionales los hace bastante flexibles. De hecho, las longitudes de las filas en el arreglo `b` no tienen que ser iguales. Por ejemplo,

```
int b[][] = { { 1, 2 }, { 3, 4, 5 } };
```

crea el arreglo entero `b` con dos elementos (los cuales se determinan según el número de inicializadores de arreglos anidados) que representan las filas del arreglo bidimensional. Cada elemento de `b` es una referencia a un arreglo unidimensional de variables `int`. El arreglo `int` de la fila `0` es un arreglo unidimensional con dos elementos (`1` y `2`), y el arreglo `int` de la fila `1` es un arreglo unidimensional con tres elementos (`3`, `4` y `5`).

Creación de arreglos bidimensionales mediante expresiones de creación de arreglos

Un arreglo multidimensional con el mismo número de columnas en cada fila puede crearse mediante una expresión de creación de arreglos. Por ejemplo, en las siguientes líneas se declara el arreglo `b` y se le asigna una referencia a un arreglo de tres por cuatro:

```
int b[][];
b = new int[ 3 ][ 4 ];
```

En este caso utilizamos los valores literales `3` y `4` para especificar el número de filas y columnas, respectivamente, pero esto no se requiere. Los programas también pueden utilizar variables para especificar las dimensiones de los arreglos. Al igual que con los arreglos unidimensionales, los elementos de un arreglo multidimensional se inicializan cuando se crea el objeto arreglo.

Un arreglo multidimensional, en el que cada fila tiene un número distinto de columnas, puede crearse de la siguiente manera:

```
int b[][];
b = new int[ 2 ][ ];          // crear 2 filas
b[ 0 ] = new int[ 5 ];        // crear 5 columnas para la fila 0
b[ 1 ] = new int[ 3 ];        // crear 3 columnas para la fila 1
```

Estas instrucciones crean un arreglo bidimensional con dos filas. La fila `0` tiene cinco columnas y la fila `1` tiene 3 columnas.

Ejemplo de arreglos bidimensionales: cómo mostrar los valores de los elementos

El applet de la figura 7.14 demuestra cómo inicializar arreglos bidimensionales con inicializadores de arreglos, y cómo utilizar ciclos `for` anidados (líneas 31 a 38) para *recorrer* los arreglos (es decir, manipular cada uno de los elementos de cada arreglo).

```
1  // Fig. 7.14: InicArreglo.java
2  // Inicialización de arreglos bidimensionales.
3  import java.awt.Container;
4  import javax.swing.*;
5
6  public class InicArreglo extends JApplet {
7     JTextArea areaSalida;
8
9     // configurar la GUI e inicializar el applet
10    public void init()
11    {
12       areaSalida = new JTextArea();
13       Container contenedor = getContentPane();
14       contenedor.add( areaSalida );
15
16       int arreglo1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
17       int arreglo2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
18
19       areaSalida.setText( "Los valores en arreglo1 por fila son\n" );
20       crearSalida( arreglo1 );
21
22       areaSalida.append( "\nLos valores en arreglo2 por fila son\n" );
23       crearSalida( arreglo2 );
24
25    } // fin del método init
26
27    // anexar filas y columnas de un arreglo a areaSalida
28    public void crearSalida( int arreglo[][] )
29    {
30       // iterar a través de las filas del arreglo
31       for ( int fila = 0; fila < arreglo.length; fila++ ) {
32
33          // iterar a través de las columnas de la fila actual
34          for ( int columna = 0; columna < arreglo[ fila ].length; columna++ )
35             areaSalida.append( arreglo[ fila ][ columna ] + " " );
36
37          areaSalida.append( "\n" );
38       }
39
40    } // fin del método crearSalida
41
42 } // fin de la clase InicArreglo
```

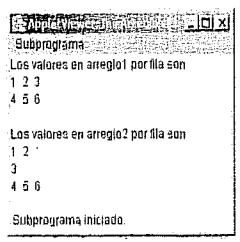


Figura 7.14 Inicialización de arreglos bidimensionales.

http://librosysolucionarios.net

El programa declara dos arreglos en el método `init`. En la declaración de `arreglo1` (línea 16) se utilizan inicializadores de arreglos anidados para inicializar la primera fila del arreglo con los valores `1`, `2` y `3`, y la segunda fila con los valores `4`, `5` y `6`. En la declaración de `arreglo2` (línea 17) se utilizan inicializadores anidados de distintas longitudes. En este caso, la primera fila se inicializa para tener dos elementos con los valores `1` y `2`, respectivamente; la segunda fila, para tener un elemento con el valor `3`; y la tercera fila, para tener tres elementos con los valores `4`, `5` y `6`, respectivamente.

En la línea 20 del método `init` se hace una llamada al método `crearSalida` (declarado en las líneas 28 a 40) para anexar los elementos de cada arreglo al área de texto `areaSalida`. El método `crearSalida` especifica el parámetro arreglo como `int arreglo[][]` para indicar que el método recibe un arreglo multidimensional como argumento. La instrucción `for` anidada (líneas 31 a 38) muestra las filas de un arreglo bidimensional. En la instrucción `for` externa, la expresión `arreglo.length` determina el número de filas en el arreglo. En la instrucción `for` interna, la expresión `arreglo[ fila ].length` determina el número de columnas en la fila actual del arreglo. Esta condición permite al ciclo determinar el número exacto de columnas en cada fila.

Manipulaciones comunes en arreglos multidimensionales, realizadas mediante instrucciones `for`

En muchas manipulaciones comunes en arreglos se utilizan instrucciones `for`. Como ejemplo, la siguiente instrucción `for` asigna a todos los elementos en la tercera fila del arreglo `a`, en la figura 7.13, el valor de cero:

```
for ( int columna = 0; columna < a[ 2 ].length; columna++ )
   a[ 2 ][ columna ] = 0;
```

Aquí especificamos la *tercera* fila; por lo tanto, sabemos que el primer índice siempre será `2` (`0` es la primera fila y `1` es la segunda). Este ciclo `for` varía solamente el segundo índice (es decir, el índice de la columna). La instrucción `for` anterior es equivalente a las siguientes instrucciones de asignación:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente instrucción `for` anidada suma el total de los valores de todos los elementos del arreglo `a`:

```
int total = 0;

for ( int fila = 0; fila < a.length; fila++ )

   for ( int columna = 0; columna < a[ fila ].length; columna++ )

      total += a[ fila ][ columna ];
```

Esta instrucción `for` suma el total de los elementos del arreglo, una fila a la vez. La instrucción `for` externa empieza asignando `0` al índice `fila`, de manera que los elementos de la primera fila puedan totalizarse mediante la instrucción `for` interna. Después, la instrucción `for` externa incrementa `fila` a `1`, de manera que la segunda fila pueda totalizarse. Luego, la instrucción `for` externa incrementa `fila` a `2`, para que la tercera fila pueda totalizarse. El resultado puede mostrarse al terminar la instrucción `for` anidada.

Ejemplo de arreglo bidimensional: resumen de las calificaciones de los exámenes de los estudiantes

El applet de la figura 7.15 realiza otras manipulaciones comunes en arreglos, en el arreglo de tres por cuatro llamado `calificaciones`. Cada fila del arreglo representa a un estudiante, y cada columna representa una calificación en uno de los cuatro exámenes que tomaron los estudiantes durante el semestre. Cuatro métodos se encargan de las manipulaciones en el arreglo. El método `minima` (líneas 48 a 65) determina la calificación más baja de cualquier estudiante durante el semestre. El método `maxima` (líneas 68 a 85) determina la calificación más alta de cualquier estudiante durante el semestre. El método `promedio` (líneas 88 a 99) determina el promedio semestral de un estudiante específico. El método `crearCadena` (líneas 102 a 118) anexa el arreglo bidimensional a la cadena `salida`, en formato tabular.

```
1  // Fig. 7.15: ArregloDoble.java
2  // Ejemplo de arreglo bidimensional.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class ArregloDoble extends JApplet {
7     int calificaciones[][] = { { 77, 68, 86, 73 },
8                                { 96, 87, 89, 81 },
9                                { 70, 90, 86, 81 } };
10
11    int estudiantes, examenes;
12    String salida;
13    JTextArea areaSalida;
14
15    // inicializar campos
16    public void init()
17    {
18       estudiantes = calificaciones.length;      // número de estudiantes
19       examenes = calificaciones[ 0 ].length;    // número de exámenes
20
21       // crear objeto JTextArea y adjuntarlo al applet
22       areaSalida = new JTextArea();
23       Container contenedor = getContentPane();
24       contenedor.add( areaSalida );
25
26       // crear cadena de salida
27       salida = "El arreglo es:\n";
28       crearCadena();
29
30       // llamar a los métodos minima y maxima
31       salida += "\n\nCalificación más baja: " + minima() +
32          "\nCalificación más alta: " + maxima() + "\n";
33
34       // llamar al método promedio para calcular el promedio de cada estudiante
35       for ( int contador = 0; contador < estudiantes; contador++ )
36          salida += "\nEl promedio del estudiante " + contador + " es " +
37             promedio( calificaciones[ contador ] ); // pasar fila arreglo
               calificaciones
38
39       // cambiar el tipo de letra a mostrar de areaSalida
40       areaSalida.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
41
42       // colocar la cadena salida en areaSalida
43       areaSalida.setText( salida );
44
45    } // fin del método init
46
47    // buscar calificación mínima
48    public int minima()
49    {
50       // suponer que el primer elemento del arreglo calificaciones es el menor
51       int califBaja = calificaciones[ 0 ][ 0 ];
52
53       // iterar a través de las filas del arreglo calificaciones
```

Figura 7.15 Arreglos bidimensionales. (Parte 1 de 3.)


```
54       for ( int fila = 0; fila < estudiantes; fila++ )
55
56          // iterar a través de las columnas de la fila actual
57          for ( int columna = 0; columna < examenes; columna++ )
58
59             // si la calificación es menor que califBaja, asignarla a califBaja
60             if ( calificaciones[ fila ][ columna ] < califBaja )
61                califBaja = calificaciones[ fila ][ columna ];
62
63       return califBaja;  // devolver la calificación más baja
64
65    } // fin del método minima
66
67    // buscar calificación máxima
68    public int maxima()
69    {
70       // suponer que el primer elemento del arreglo calificaciones es el mayor
71       int califAlta = calificaciones[ 0 ][ 0 ];
72
73       // iterar a través de las filas del arreglo calificaciones
74       for ( int fila = 0; fila < estudiantes; fila++ )
75
76          // iterar a través de las columnas de la fila actual
77          for ( int columna = 0; columna < examenes; columna++ )
78
79             // si la calificación es mayor que califAlta, asignarla a califAlta
80             if ( calificaciones[ fila ][ columna ] > califAlta )
81                califAlta = calificaciones[ fila ][ columna ];
82
83       return califAlta;  // devolver la calificación más alta
84
85    } // fin del método maxima
86
87    // determinar la calificación promedio para un estudiante (o conjunto de calif.)
88    public double promedio( int conjuntoDeCalificaciones[] )
89    {
90       int total = 0;  // inicializar el total
91
92       // sumar calificaciones para un estudiante
93       for ( int cuenta = 0; cuenta < conjuntoDeCalificaciones.length; cuenta++ )
94          total += conjuntoDeCalificaciones[ cuenta ];
95
96       // devolver el promedio de calificaciones
97       return ( double ) total / conjuntoDeCalificaciones.length;
98
99    } // fin del método promedio
100
101   // crear cadena de salida
102   public void crearCadena()
103   {
104      salida += "            ";  // se usa para alinear los encabezados de columnas
105
106      // crear encabezados de columnas
107      for ( int contador = 0; contador < examenes; contador++ )
```

Figura 7.15 Arreglos bidimensionales. (Parte 2 de 3.)

http://librosysolucionarios.net

```
108            salida += "[" + contador + "]   ";
109
110         // crear filas/columnas de texto que representan el arreglo calificaciones
111         for ( int fila = 0; fila < estudiantes; fila++ ) {
112            salida += "\ncalificaciones[" + fila + "]    ";
113
114            for ( int columna = 0; columna < examenes; columna++ )
115               salida += calificaciones[ fila ][ columna ] + "    ";
116         }
117
118      } // fin del método crearCadena
119
120   } // fin de la clase ArregloDoble
```

```
Subprograma
El arreglo es:
                [0]  [1]  [2]  [3]
calificaciones[0]  77   68   86   73
calificaciones[1]  96   87   89   81
calificaciones[2]  70   90   86   81

Calificación más baja: 68
Calificación más alta: 96

El promedio del estudiante 0 es 76.0
El promedio del estudiante 1 es 88.25
El promedio del estudiante 2 es 81.75

Subprograma iniciado.
```

Figura 7.15 Arreglos bidimensionales. (Parte 3 de 3.)

Los métodos `minima`, `maxima`, y `crearCadena` utilizan el arreglo `calificaciones`, junto con las variables `estudiantes` (número de filas en el arreglo) y `examenes` (número de columnas en el arreglo). Cada método itera a través del arreglo `calificaciones` mediante el uso de instrucciones `for` anidadas; por ejemplo, la instrucción `for` anidada de la declaración del método `minima` (líneas 54 a 61). La instrucción `for` externa asigna `0` a `fila` (el índice de las filas), de manera que los elementos de la primera fila puedan compararse con la variable `califBaja` en el cuerpo de la instrucción `for` interna. La instrucción `for` interna itera a través de las cuatro calificaciones de una fila específica y compara cada una de ellas con `califBaja`. Si una calificación es menor que `califBaja`, a `califBaja` se le asigna esa calificación. La instrucción `for` externa incrementa a continuación el índice de fila en `1`, y los elementos de la segunda fila se comparan con la variable `califBaja`. La instrucción `for` externa incrementa entonces el índice de fila en `2`, y los elementos de la tercera fila se comparan con la variable `califBaja`. Al completarse la ejecución de la instrucción anidada, `califBaja` contiene la calificación más baja del arreglo bidimensional. El método `maxima` funciona de manera similar al método `minima`.

El método `promedio` toma un argumento: un arreglo unidimensional de resultados de las pruebas de un estudiante específico. Cuando en la línea 37 se hace la llamada a `promedio`, el argumento es `calificaciones[ contador ]`, el cual especifica que se va a pasar a `promedio` una fila específica del arreglo `calificaciones`. Por ejemplo, el argumento `calificaciones[ 1 ]` representa los cuatro valores (un arreglo unidimensional de calificaciones) almacenados en la segunda fila del arreglo bidimensional `calificaciones`. Recuerde que en Java un arreglo bidimensional es un arreglo con elementos que son arreglos unidimensionales. El método `promedio` calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de las pruebas y devuelve el resultado de punto flotante como un valor `double` (línea 97).

http://librosysolucionarios.net

7.10 (Ejemplo práctico opcional) Acerca de los objetos: Colaboración entre los objetos

En esta sección nos concentraremos en las colaboraciones (interacciones) entre los objetos. Cuando dos objetos se comunican entre sí para realizar una tarea, se dice que *colaboran* (para ello, un objeto invoca a las operaciones del otro). Decimos que los objetos envían *mensajes* a otros objetos. Una *colaboración* consiste en que un objeto de una clase envía un mensaje a un objeto de otra clase. En la sección 11.9 explicaremos cómo se envían y reciben los mensajes en Java.

El mensaje enviado por el primer objeto invoca a una operación del segundo objeto. En la sección 6.15 determinamos muchas de las operaciones de las clases en nuestro sistema. En esta sección, nos concentraremos en los mensajes que invocan a esas operaciones. La figura 7.16 es la tabla de clases y frases verbales de la sección 6.15. Hemos eliminado todas las frases verbales en las clases `Elevador` y `Persona` que no corresponden a las operaciones. Las frases restantes son nuestra primera estimación de las colaboraciones en nuestro sistema. A medida que avancemos por esta sección y las siguientes secciones "Acerca de los objetos" que quedan en el libro, descubriremos relaciones adicionales.

Examinaremos la lista de frases verbales para determinar las colaboraciones en nuestro sistema. Por ejemplo, la clase `Elevador` contiene la frase "restablece el botón del elevador". Para realizar esta tarea, un objeto de la clase `Elevador` envía el mensaje `restablecerBoton` a un objeto de la clase `BotonElevador` (invocando a la operación `restablecerBoton` de `BotonElevador`). En la figura 7.17 se muestran todas las colaboraciones que pueden obtenerse de nuestra tabla de frases verbales. De acuerdo con la figura 7.16, el `Elevador`[5] suena el `Timbre` y cierra (y abre) la `PuertaElevador`, por lo que incluimos los mensajes `sonarTimbre`, `abrirPuerta` y `cerrarPuerta` en la figura 7.17. Sin embargo, debemos considerar cómo se abren y cierran los objetos `PuertaPiso`. De acuerdo con el diagrama de clases de la figura 3.19, `ConductoElevador` se asocia con `PuertaPiso`. El `Elevador` indica su llegada (suponemos que al `ConductoElevador`) enviando un mensaje `elevadorLlego`. El `ConductoElevador` responde a este mensaje restableciendo el `BotonPiso` apropiado y encendiendo la `Luz` apropiada (el `ConductoElevador` envía los mensajes `restablecerBoton` y `encenderLuz`). En este punto del diseño, podemos suponer que el `Elevador` también indica su partida (es decir, el `Elevador` envía un mensaje `elevadorPartio` al `ConductoElevador`, el cual a su vez apaga la `Luz` apropiada enviando un mensaje `apagarLuz`).

Una `Persona` puede oprimir un `BotonPiso` o el `BotonElevador`. Una `Persona` puede entrar o salir del `Elevador`. Por lo tanto, una `Persona` puede enviar mensajes `oprimirBoton`, `entrarEleva-`

Clase	Frases verbales
`Elevador`	Restablece el botón del elevador, suena el timbre del elevador, indica su llegada, indica su partida, abre su puerta, cierra su puerta.
`ConductoElevador`	Apaga la luz, enciende la luz, restablece el botón del piso.
`Persona`	Oprime el botón del piso, oprime el botón del elevador, viaja en el elevador, entra al elevador, sale del elevador.
`BotonPiso`	Llama (solicita) el elevador.
`BotonElevador`	Indica al elevador que vaya al piso opuesto.
`PuertaPiso`	Indica a la persona que entre al elevador (al abrirse).
`PuertaElevador`	Indica a la persona que salga del elevador (al abrirse), abre la puerta del piso, cierra la puerta del piso.

Figura 7.16 Frases verbales para cada clase que exhibe comportamientos en el simulador.

5. Nos referimos a un objeto utilizando el nombre de la clase de ese objeto, anteponiendo un artículo ("un", "una", "el" o "la"); por ejemplo, el `Elevador` se refiere a un objeto de la clase `Elevador`. Nuestra sintaxis es una informalidad más concisa (evitamos repetir la frase "un objeto de la clase...").

Un objeto de la clase...	Envía el mensaje...	A un objeto de la clase...
Elevador	restablecerBoton	BotonElevador
	sonarTimbre	Timbre
	elevadorLlego	ConductoElevador
	elevadorPartio	ConductoElevador
	abrirPuerta	PuertaElevador
	cerrarPuerta	PuertaElevador
ConductoElevador	restablecerBoton	BotonPiso
	encenderLuz	Luz
	apagarLuz	Luz
Persona	oprimirBoton	BotonPiso, BotonElevador
	entrarElevador	Elevador
	salirElevador	Elevador
BotonPiso	solicitarElevador	Elevador
BotonElevador	moverElevador	Elevador
PuertaPiso	puertaAbierta	Persona
	puertaCerrada	Persona
PuertaElevador	puertaAbierta	Persona
	puertaCerrada	Persona
	abrirPuerta	PuertaPiso
	cerrarPuerta	PuertaPiso

Figura 7.17 Colaboraciones en el sistema del elevador.

dor y **salirElevador**. Un **BotonPiso** solicita (o llama) al **Elevador**, por lo que un **BotonPiso** puede enviar un mensaje **solicitarElevador**. El **BotonElevador** indica al **Elevador** que empiece a moverse hacia el otro piso, por lo que el **BotonElevador** puede enviar un mensaje **moverElevador**.

Tanto un **BotonPiso** como la **PuertaElevador** informan a una **Persona** que las puertas se han abierto o cerrado, por lo que ambos objetos envían mensajes **puertaAbierta** o **puertaCerrada**.[6] Por último, la **PuertaElevador** debe enviar mensajes **abrirPuerta** y **cerrarPuerta** a un **BotonPiso** para garantizar que estas puertas se abran y cierren al mismo tiempo.

Diagramas de colaboración

Ahora consideraremos los objetos que deben interactuar de manera que las personas en nuestra simulación puedan entrar y salir del elevador, cuando éste llegue a un piso. UML cuenta con el *diagrama de colaboración* para modelar dichas interacciones. Estos diagramas son un tipo de *diagrama de interacción*; modelan los aspectos de un sistema relacionados con los comportamientos, al proporcionar información acerca de cómo interactúan los objetos. Los diagramas de colaboración enfatizan qué objetos participan en las interacciones. (El otro tipo de diagrama de interacción es el *diagrama de secuencia*, que presentaremos en el capítulo 16.) En la figura 7.18 se muestra un diagrama de colaboración que modela a una persona que oprime el botón de un piso. Los objetos se modelan con nombres de la forma **nombreObjeto : NombreClase**. En este ejemplo descartamos el nombre del objeto, ya que sólo nos interesa el tipo del objeto. Los objetos colaboradores se conectan con líneas continuas, y los mensajes se pasan entre objetos a lo largo de estas líneas, en la dirección que indican las flechas. El nombre del mensaje, que aparece enseguida de la flecha, es el nombre de un método que pertenece al

6. Observe que la mayoría de los mensajes realizan alguna acción específica sobre el objeto receptor; por ejemplo, el **Elevador** restablece el **BotonElevador**. Sin embargo, otros mensajes informan a los objetos receptores sobre *eventos* que ya han ocurrido; por ejemplo, la **PuertaPiso** informa a la **Persona** que la **PuertaPiso** se ha abierto. En la sección 16.9 hablaremos más sobre el tema de los eventos; por ahora procederemos como si los dos tipos de mensajes no tuvieran distinción.

objeto receptor; piense en el nombre como un "servicio" que proporciona el objeto receptor para sus objetos emisores (sus "clientes").

La flecha en la figura 7.18 representa un mensaje en el UML y la llamada a un método (o *llamada síncrónica*) en Java. Esta flecha indica que el flujo de control va desde el objeto emisor (una **Persona**) hacia el objeto receptor (un **BotonPiso**). Ya que ésta es una llamada síncrónica, tal vez el objeto emisor no envíe otro mensaje sino hasta que el objeto receptor procese ese mensaje y devuelva el control al objeto emisor. Por ejemplo, en la figura 7.18 una **Persona** llama al método **oprimirBoton** de un **BotonPiso** y tal vez no envíe otro mensaje a un objeto, sino hasta que **oprimirBoton** haya terminado y devuelva el control a esa **Persona**. Si nuestro programa contiene un objeto **BotonPiso** llamado **botonPrimerPiso**, y si suponemos que la **Persona** administra el flujo de control, la implementación en código de Java en la clase **Persona** que representara a este diagrama de colaboración sería:

```
botonPrimerPiso.oprimirBoton();
```

En la figura 7.19 se muestra un diagrama de colaboración que modela las interacciones entre objetos en el sistema, mientras objetos de la clase **Persona** entran y salen del elevador. La colaboración empieza cuando el **Elevador** llega a un **Piso**. El número a la izquierda del nombre del mensaje indica el orden en el que éste se pasa. La *secuencia de mensajes* en un diagrama de colaboración progresa en orden numérico, de menor a

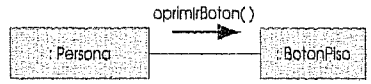


Figura 7.18 Diagrama de colaboración de una persona que oprime el botón de un piso.

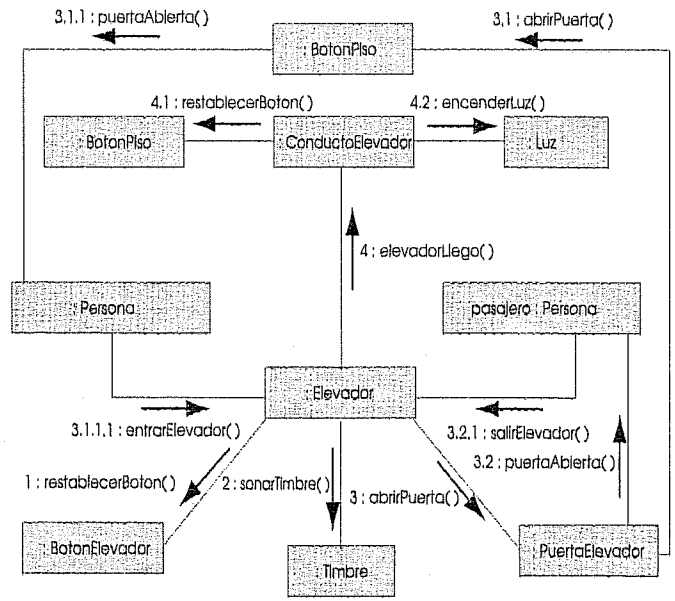


Figura 7.19 Diagrama de colaboración para los pasajeros que salen y entran al elevador.

http://librosysolucionarios.net

mayor. En este diagrama, la numeración empieza con el mensaje 1 y termina con el mensaje 4.2. La secuencia de pasar mensajes sigue una estructura anidada; por ejemplo, el mensaje 3.1 es el primer mensaje anidado en el mensaje 3, y mensaje 3.2 es el segundo mensaje anidado en el mensaje 3. El mensaje 3.2.1 es el primer mensaje anidado en el mensaje 3.2. Un mensaje podrá pasarse solamente cuando se hayan pasado todos los mensajes anidados del mensaje anterior. Por ejemplo, en la figura 7.19, el `Elevador` pasa el mensaje 4 después de haber pasado los mensajes 3, 3.1, 3.1.1, 3.1.1.1, 3.2 y 3.2.1, en ese orden.

El `Elevador` envía el mensaje `restablecerBoton` (mensaje 1) al `BotonElevador` para restablecerlo. A continuación, el `Elevador` envía el mensaje `sonarTimbre` (mensaje 2) al `Timbre`, después abre la `PuertaElevador` enviándole el mensaje `abrirPuerta` (mensaje 3). Después la `PuertaElevador` abre la `PuertaPiso` enviándole un mensaje `abrirPuerta` (mensaje 3.1, en la parte superior del diagrama) a esa `PuertaPiso`. La `PuertaPiso` informa a la `Persona` en espera que se ha abierto la `PuertaPiso` (mensaje 3.1.1), y la `Persona` en espera entra al `Elevador` (mensaje 3.1.1.1). Luego, la `PuertaElevador` informa al `pasajero` que se ha abierto (mensaje 3.2), para que el `pasajero` pueda salir del `Elevador` (mensaje 3.2.1). Por último, el `Elevador` informa al `ConductoElevador` de su llegada (mensaje 4), de manera que el `ConductoElevador` pueda restablecer el `BotonPiso` (mensaje 4.1) y encender la `Luz` (mensaje 4.2).

Desafortunadamente, este diseño crea un problema. De acuerdo con el diagrama, la `Persona` en espera entra al `Elevador` (mensaje 3.1.1.1) antes de que el `pasajero` (mensaje 3.2.1) salga. En la sección 11.9 modificaremos este diagrama, para indicar con más precisión el paso de mensajes cuando hablemos sobre el *manejo de eventos*. En la sección 16.11 refinaremos aún más este diagrama y aplicaremos el subprocesamiento múltiple, la sincronización y las clases activas en nuestro sistema, para forzar a que la `Persona` espere a que el `pasajero` salga del `Elevador`.

RESUMEN

- Un arreglo es un objeto que contiene elementos (componentes), los cuales tienen todos el mismo tipo. Para hacer referencia a un elemento específico dentro del arreglo utilizamos una expresión de acceso a un arreglo, la cual contiene el nombre de la referencia a ese arreglo y el índice (o subíndice) del elemento.
- Cada arreglo tiene un campo `length`, que contiene el número de elementos del arreglo en el momento en que el programa crea el objeto arreglo.
- Un índice puede ser un entero o una expresión entera (pero no de tipo `long`). Si un programa utiliza una expresión como índice, el programa evalúa la expresión para determinar el elemento específico del arreglo.
- En Java, los arreglos siempre empiezan con el elemento cero. Por lo tanto, es importante observar la diferencia entre el "séptimo elemento del arreglo" y el "elemento siete del arreglo". El séptimo elemento tiene un índice de 6, mientras que el elemento siete del arreglo tiene un índice de 7 (el octavo elemento del arreglo).
- En Java, los arreglos son objetos. Una expresión de creación de arreglos, o inicializador de arreglos, reserva espacio para un arreglo. La siguiente expresión crea un arreglo de 100 valores `int`:

```
int b[] = new int[ 100 ];
```

- Al declarar un arreglo, el tipo de éste y los corchetes pueden combinarse al principio de la declaración, para indicar que todos los identificadores en la declaración representan arreglos, como en:

```
double[] arreglo1, arreglo2;
```

- Los elementos de un arreglo pueden inicializarse con listas inicializadoras en una declaración, o mediante una asignación.
- Java impide las referencias a elementos que estén más allá de los límites de un arreglo. Si esto ocurre durante la ejecución de un programa, se produce una excepción `ArrayIndexOutOfBoundsException`.
- Las constantes deben recibir un valor antes de poder usarse, y no pueden modificarse a partir de entonces.
- Para pasar un arreglo a un método se especifica el nombre del arreglo, sin corchetes. Para pasar un elemento individual de un arreglo a un método se utiliza una expresión de acceso a arreglos, la cual contiene el nombre de la referencia al arreglo y el índice (o índices) apropiado.
- Los arreglos se pasan a los métodos como referencias; por lo tanto, los métodos llamados pueden modificar los valores de los elementos en los arreglos originales del método que hace la llamada. Los elementos individuales de arreglos de tipo primitivo se pasan a los métodos por valor.
- Para aceptar un arreglo como argumento para un método, la lista de parámetros de ese método debe declarar un arreglo como parámetro.

- Un arreglo puede ordenarse mediante la técnica del ordenamiento burbuja. Se realizan varias pasadas por el arreglo. En cada pasada se comparan pares de elementos sucesivos. Si un par está en orden (o los valores son idénticos), se deja como está. Si un par no está en orden, los valores se intercambian.
- Una búsqueda lineal compara cada elemento del arreglo con una clave de búsqueda. Si el arreglo no se encuentra en un orden específico, es igual de probable que el valor se encuentre en el primer elemento, a que se encuentre en el segundo. Por lo tanto, en promedio el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del arreglo. La búsqueda lineal funciona bien para arreglos pequeños y es aceptable incluso para arreglos grandes y desordenados.
- Para los arreglos ordenados, la búsqueda binaria descarta de la búsqueda a la mitad de los elementos en el arreglo, después de cada comparación. El algoritmo localiza el elemento medio del arreglo y lo compara con la clave de búsqueda. Si son iguales significa que se ha encontrado la clave de búsqueda, y se devuelve el índice de ese elemento en el arreglo. En caso contrario, el problema se reduce a buscar en la mitad del arreglo que aún se está considerando para la búsqueda.
- La mayoría de los componentes de la GUI tienen el método `setFont` para cambiar el tipo de letra del texto en el componente de la GUI. El método requiere un objeto `Font` (paquete `java.awt`) como argumento.
- Un objeto `Font` se inicializa con tres argumentos: un `String` que representa el nombre del tipo de letra, un `int` que representa el estilo de letra y un `int` que representa el tamaño del tipo de letra. El estilo puede ser `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` o `Font.BOLD+Font.ITALIC`. El tamaño de punto representa el tamaño del tipo de letra. Hay 72 puntos en una pulgada. El tamaño actual del texto en la pantalla puede variar, con base en el tamaño y la resolución de la pantalla.
- Los arreglos pueden utilizarse para representar tablas de valores que consisten de información ordenada en filas y columnas. Para identificar un elemento específico de una tabla, se especifican dos índices: el primero identifica la fila en la que se encuentra el elemento, y el segundo identifica la columna. Las tablas o arreglos que requieren dos índices para identificar a un elemento específico se conocen como arreglos bidimensionales.
- Los arreglos multidimensionales en Java se mantienen como arreglos de arreglos.
- Un arreglo bidimensional puede inicializarse con un inicializador de arreglos, de la siguiente manera:

tipoArreglo nombreArreglo[][] = { { *inicializador fila1* }, { *inicializador fila2* }, ... };

- Para crear un arreglo con un número fijo de filas y columnas, se utiliza:

tipoArreglo nombreArreglo[][] = new *tipoArreglo*[*numFilas*][*numColumnas*];

- Para pasar una fila de un arreglo bidimensional a un método que reciba un arreglo unidimensional, simplemente se pasa el nombre de la referencia al arreglo, seguido solamente por el índice de la fila.

TERMINOLOGÍA

`a[i]`
`a[i][j]`
área temporal para intercambiar valores
arreglo
arreglo bidimensional
arreglo de *m* por *n*
arreglo multidimensional
arreglo unidimensional
buscar en un arreglo
búsqueda binaria de un arreglo
búsqueda lineal de un arreglo
clave de búsqueda
componente de un arreglo
comprobación de límites
constante
constante con nombre
corchetes, `[]`
declarar un arreglo
elemento cero
error de desplazamiento por uno
expresión de acceso a un arreglo
expresión de creación de arreglos
`final`
`Font`, clase de `java.awt`
`Font.BOLD`
`Font.ITALIC`
`Font.PLAIN`
formato tabular
índice
índice de columna
índice de fila
inicializador de arreglos
inicializar un arreglo
método de hundimiento
nombre de un arreglo
número de posición
ordenamiento
ordenamiento burbuja
ordenar un arreglo
pasada de un método burbuja
paso de arreglos a métodos
paso por referencia
paso por valor
`setFont`, método
subíndice
tabla de valores
valor de un elemento

EJERCICIOS DE AUTOEVALUACIÓN

7.1 Complete las siguientes oraciones:

a) Las listas y tablas de valores pueden guardarse en _____.

b) Los elementos de un arreglo se relacionan por el hecho de que tienen el mismo _____ y _____.

http://librosysolucionarios.net

c) El número utilizado para referirse a un elemento específico de un arreglo se conoce como el _____ de ese elemento.
d) El proceso de colocar los elementos de un arreglo en orden se conoce como _____.
e) Al proceso de determinar si un arreglo contiene cierto valor clave se le conoce como _____ en el arreglo.
f) Un arreglo que utiliza dos índices se conoce como un arreglo _____.

7.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Un arreglo puede guardar muchos tipos distintos de valores.
b) El índice de un arreglo debe ser generalmente de tipo `float`.
c) Un elemento individual de un arreglo que se pasa a un método y se modifica ahí mismo, contendrá el valor modificado cuando el método llamado termine su ejecución.

7.3 Realice las siguientes tareas para un arreglo llamado `fracciones`:
a) Declare una constante llamada `TAMANIO_ARREGLO` que se inicialice con `10`.
b) Declare un arreglo con `TAMANIO_ARREGLO` elementos de tipo `float`, e inicialice los elementos con `0`.
c) Nombre el cuarto elemento del arreglo.
d) Haga referencia al elemento cuatro del arreglo.
e) Asigne el valor `1.667` del elemento nueve del arreglo.
f) Asigne el valor `3.333` al séptimo elemento del arreglo.
g) Sume todos los elementos del arreglo, utilizando una instrucción `for`. Declare la variable entera `x` como variable de control para el ciclo.

7.4 Realice las siguientes tareas para un arreglo llamado `tabla`:
a) Declare y cree el arreglo como un arreglo entero con tres filas y tres columnas. Suponga que se ha declarado la constante `TAMANIO_ARREGLO` con el valor de `3`.
b) ¿Cuántos elementos contiene el arreglo?
c) Utilice una instrucción `for` para inicializar cada elemento del arreglo con la suma de sus índices. Suponga que se declaran las variables enteras `x` e `y` como variables de control.

7.5 Encuentre y corrija el error en cada uno de los siguientes fragmentos de programa:
a)
```
final int TAMANIO_ARREGLO = 5;
TAMANIO_ARREGLO = 10;
```
b)
```
Suponga que int b[] = new int[ 10 ];
for ( int i = 0; i <= b.length; i++ )
   b[ i ] = 1;
```
c)
```
Suponga que int a[][] = { { 1, 2 }, { 3, 4 } };
   a[ 1, 1 ] = 5;
```

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

7.1 a) Arreglos. b) Nombre, tipo. c) Índice (o subíndice, o número de posición). d) Ordenamiento. e) Buscar. f) Bidimensional.

7.2 a) Falso. Un arreglo sólo puede guardar valores del mismo tipo.
b) Falso. El índice de un arreglo debe ser un entero o una expresión entera.
c) Para los elementos individuales de tipo primitivo en un arreglo: falso. Dichos elementos se pasan por valor. Si se pasa una referencia a un arreglo, entonces las modificaciones a los elementos del arreglo se reflejan en el original. Para los elementos individuales de un tipo no primitivo: verdadero. Dichos elementos se pasan por referencia, y los cambios en el objeto se reflejarán en el elemento original del arreglo.

7.3 a)
```
final int TAMANIO_ARREGLO = 10;
```
b)
```
float fracciones[] = new float[ TAMANIO_ARREGLO ];
```
c)
```
fracciones[ 3 ]
```
d)
```
fracciones[ 4 ]
```
e)
```
fracciones[ 9 ] = 1.667;
```
f)
```
fracciones[ 6 ] = 3.333;
```
g)
```
float total = 0.0;
for ( int x = 0; x < fracciones.length; x++ )
   total += fracciones[ x ];
```

7.4 a)
```
int tabla[][] = new int[ TAMANIO_ARREGLO ][ TAMANIO_ARREGLO ];
```
b) Nueve.

http://librosysolucionarios.net

c)
```
for ( int x = 0; x < tabla.length; x++ )
   for ( int y = 0; y < tabla[ x ].length; y++ )
      tabla[ x ][ y ] = x + y;
```

7.5 a) Error: Se trata de asignar un valor a una constante después de inicializarla.
Corrección: Asigne el valor correcto a la constante en una declaración `final int TAMANIO_ARREGLO`, o crear otra variable.
b) Error: Se está haciendo referencia al elemento de un arreglo que está fuera de los límites del arreglo (`b[10]`).
Corrección: Cambie el operador `<=` por `<`.
c) Error: La indización del arreglo se está realizando en forma incorrecta.
Corrección: Cambie la instrucción por `a[ 1 ][ 1 ] = 5;`.

EJERCICIOS

7.6 Complete las siguientes oraciones:
a) Java guarda listas de valores en _____.
b) Los elementos de un arreglo se relacionan por el hecho de que _____.
c) Al referirse al elemento de un arreglo, el número de posición contenido entre corchetes se conoce como _____.
d) Los nombres de los cuatro elementos de un arreglo unidimensional llamado `p` son _____, _____, _____ y _____.
e) Al proceso de nombrar un arreglo, declarar su tipo y especificar el número de dimensiones se le conoce como _____ el arreglo.
f) El proceso de colocar los elementos de un arreglo, ya sea en orden ascendente o descendente, se conoce como _____.
g) En un arreglo bidimensional, el primer índice identifica el(la) _____ de un elemento y el segundo índice identifica el(la) _____ de un elemento.
h) Un arreglo de *m* por *n* contiene _____ filas, _____ columnas y _____ elementos.
i) El nombre del elemento en la fila 3 y la columna 5 del arreglo `d` es _____.

7.7 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Para referirse a una ubicación o elemento específico dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento específico.
b) Al declarar un arreglo, se reserva espacio para el mismo.
c) Para indicar que deben reservarse 100 ubicaciones para el arreglo entero `p`, el programador escribe la declaración
```
p[ 100 ];
```
d) Un programa en Java que inicializa con cero los elementos de un arreglo con 15 elementos debe contener al menos una instrucción `for`.
e) Un programa en Java que sume el total de los elementos de un arreglo bidimensional debe contener instrucciones `for` anidadas.

7.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:
a) Mostrar el valor del séptimo elemento del arreglo tipo carácter `f`.
b) Inicializar con `8` cada uno de los cinco elementos del arreglo entero unidimensional `g`.
c) Sumar el total de los 100 elementos del arreglo `c` de punto flotante.
d) Copiar el arreglo `a` de 11 elementos en la primera porción del arreglo `b`, el cual contiene 34 elementos.
e) Determinar e imprimir los valores menor y mayor contenidos en el arreglo `w` con 99 elementos de punto flotante.

7.9 Considere un arreglo entero `t` de dos por tres.
a) Escriba una instrucción que declare y cree a `t`.
b) ¿Cuántas filas tiene `t`?
c) ¿Cuántas columnas tiene `t`?
d) ¿Cuántos elementos tiene `t`?
e) Escriba los nombres de todos los elementos en la segunda fila de `t`.
f) Escriba los nombres de todos los elementos en la tercera columna de `t`.
g) Escriba una sola instrucción que asigne cero al elemento de `t` en la fila 1 y la columna 2.

h) Escriba una serie de instrucciones que inicialice cada elemento de `t` con cero. No utilice una instrucción de repetición.
i) Escriba una instrucción `for` anidada que inicialice cada elemento de `t` con cero.
j) Escriba una instrucción `for` anidada que reciba como entrada del usuario los valores de los elementos de `t`.
k) Escriba una serie de instrucciones que determine e imprima el valor más pequeño en `t`.
l) Escriba una instrucción que muestre los elementos de la primera fila de `t`.
m) Escriba una instrucción que totalice los elementos de la tercera columna de `t`.
n) Escriba una serie de instrucciones para imprimir el contenido de `t` en un impecable formato tabular. Enliste los índices de columna como encabezados a lo largo de la parte superior, y enliste los índices de fila a la izquierda de cada fila.

7.10 Utilice un arreglo unidimensional para resolver el siguiente problema: una compañía paga a sus vendedores por comisión. Los vendedores reciben $200 por semana más el 9% de sus ventas totales de esa semana. Por ejemplo, un vendedor que acumule $5000 en ventas en una semana, recibirá $200 más el 9% de $5000, o un total de $650. Escriba un applet (utilizando un arreglo de contadores) que determine cuántos vendedores recibieron salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca a una cantidad entera):

a) $200-299
b) $300-399
c) $400-499
d) $500-599
e) $600-699
f) $700-799
g) $800-899
h) $900-999
i) $1000 en adelante

El applet debe utilizar las técnicas de la GUI introducidas en el capítulo 6. Muestre los resultados en un objeto `JTextArea`. Utilice el método `setText` de `JTextArea` para actualizar los resultados después de que el usuario introduzca cada uno de los valores.

7.11 El método de ordenamiento burbuja presentado en la figura 7.10 es ineficiente para arreglos extensos. Haga las siguientes modificaciones sencillas para mejorar el rendimiento del método burbuja:

a) Después de la primera pasada, se garantiza que el número más grande debe estar en el elemento del arreglo con el subíndice más alto; después de la segunda pasada, los dos números más grandes se encuentran "acomodados"; etcétera. En vez de realizar nueve comparaciones en cada pasada, modifique el método burbuja para hacer ocho comparaciones en la segunda pasada, siete en la tercera, etcétera.
b) Los datos en el arreglo tal vez ya se encuentren en el orden correcto, o casi en orden, entonces ¿para qué hacer nueve pasadas si bastará con menos? Modifique el ordenamiento para comprobar al final de cada pasada si se han realizado intercambios. Si no se han hecho intercambios, los datos deben encontrarse ya en el orden apropiado, por lo que el programa debe terminar. Si se han hecho intercambios, se requiere al menos de una pasada más.

7.12 Escriba instrucciones que realicen las siguientes operaciones con arreglos unidimensionales:

a) Asignar cero a los 10 elementos del arreglo `cuentas` de tipo entero.
b) Sumar uno a cada uno de los 15 elementos del arreglo `bono` de tipo entero.
c) Imprimir los cinco valores del arreglo `mejoresPuntuaciones` de tipo entero en formato de columnas.

7.13 Use un arreglo unidimensional para resolver el siguiente problema: escriba un applet que reciba como entrada 5 números, cada uno de los cuales debe estar entre 10 y 100, inclusive. A medida que se lea cada número, muéstrelo solamente si no es un duplicado de un número que ya se haya leído. Prepárese para el "peor caso", en el que los 5 números son diferentes. Use el arreglo más pequeño que sea posible para resolver este problema. El applet debe utilizar las técnicas de la GUI introducidas en el capítulo 6. Muestre los resultados en un objeto `JTextArea`. Utilice el método `setText` del método `JTextArea` para actualizar los resultados después que el usuario introduzca cada uno de los valores.

7.14 Etiquete los elementos del arreglo bidimensional `ventas` de tres por cinco, para indicar el orden en el que se establecen en cero, mediante el siguiente fragmento de programa:

```
for ( int fila = 0; fila < ventas.length; fila++ )

   for ( int col = 0; col < ventas[ fila ].length; col++ )

      ventas[ fila ][ col ] = 0;
```


	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figura 7.20 Las 36 posibles sumas de dos dados.

7.15 Escriba un applet para simular el tiro de dos dados. El programa debe utilizar `Math.random` una vez para tirar el primer dado, y de nuevo para tirar el segundo dado. Después debe calcularse la suma de los dos valores. Cada dado puede mostrar un valor entero del 1 al 6, por lo que la suma de los valores variará del 2 al 12, siendo 7 la suma más frecuente, mientras que 2 y 12 serán las sumas menos frecuentes. En la figura 7.20 se muestran las 36 posibles combinaciones de los dos dados. Su programa debe tirar los dados 36,000 veces. Utilice un arreglo unidimensional para registrar el número de veces que aparezca cada una de las posibles sumas. Muestre los resultados en un objeto `JTextArea`, en formato tabular. Además, determine si los totales son razonables (es decir, hay seis formas de tirar un 7, por lo que aproximadamente una sexta parte de los tiros deben ser 7). El applet debe utilizar las técnicas de la GUI introducidas en el capítulo 6. Incluya un objeto `JButton` para permitir que el usuario del applet tire los dados otras 36,000 veces. El applet debe restablecer los elementos del arreglo unidimensional en cero, antes de tirar de nuevo los dados.

7.16 ¿Qué hace el programa de la figura 7.21?

7.17 Escriba un programa que ejecute 1000 juegos de craps (figura 6.9) y que responda a las siguientes preguntas:

a) ¿Cuántos juegos se ganan en el primer tiro, segundo tiro, ..., vigésimo tiro y después de éste?
b) ¿Cuántos juegos se pierden en el primer tiro, segundo tiro, ..., vigésimo tiro y después de éste?
c) ¿Cuáles son las probabilidades de ganar en craps? [*Nota:* Eventualmente se dará cuenta que craps es uno de los juegos de casino más justos. ¿Qué cree usted que significa esto?]
d) ¿Cuál es la duración promedio de un juego de craps?
e) ¿Mejoran las oportunidades de ganar con la duración del juego?

7.18 (*Sistema de reservaciones de una aerolínea.*) Una pequeña aerolínea acaba de comprar una computadora para su nuevo sistema de reservaciones automatizado. Se le ha pedido a usted que programe el nuevo sistema. Usted va a escribir un applet para asignar asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su programa debe mostrar las siguientes alternativas: `Por favor escriba 1 para PrimeraClase` y `Por favor escriba 2 para Económico`. Si el usuario escribe `1`, su programa debe asignarle un asiento en la sección de primera clase (asientos 1 a 5). Si la persona escribe `2`, su programa debe asignarle un asiento en la sección económica (asientos 6 a 10). Su programa deberá entonces imprimir un pase de abordaje, indicando el número de asiento de la persona y si se encuentra en la sección de primera clase o económica del avión.

Use un arreglo unidimensional del tipo primitivo `boolean` para representar la tabla de asientos del avión. Inicialice todos los elementos del arreglo con `false` para indicar que todos los asientos están vacíos. A medida que se asigne cada asiento, establezca los elementos correspondientes del arreglo en `true` para indicar que ese asiento ya no está disponible.

Su programa nunca deberá asignar un asiento que ya haya sido asignado. Cuando esté llena la sección económica, su programa deberá preguntar a la persona si acepta ser colocada en la sección de primera clase (y viceversa). Si la persona acepta, haga la asignación de asiento apropiada. Si no acepta, imprima el mensaje `"El próximo vuelo sale en 3 horas."`

7.19 ¿Qué hace el programa de la figura 7.22?

7.20 Use un arreglo bidimensional para resolver el siguiente problema: una compañía tiene cuatro vendedores (1 a 4) que venden cinco productos distintos (1 a 5). Una vez al día, cada vendedor pasa una nota por cada tipo de producto vendido. Cada nota contiene lo siguiente:

http://librosysolucionarios.net

```
1  // Ejercicio 7.16: QueHaceEsto.java
2  import java.awt.*;
3  import javax.swing.*;
4
5  public class QueHaceEsto extends JApplet {
6     int resultado;
7
8    public void init()
9     {
10       int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
11
12       resultado = queEsEsto( arreglo, arreglo.length );
13
14       Container contenedor = getContentPane();
15       JTextArea salida = new JTextArea();
16       salida.setText( "El resultado es: " + resultado );
17       contenedor.add( salida );
18    }
19
20    public int queEsEsto( int arreglo2[], int longitud )
21    {
22       if ( longitud == 1 )
23          return arreglo2[ 0 ];
24       else
25          return arreglo2[ longitud - 1 ] + queEsEsto( arreglo2, longitud - 1 );
26    }
27 }
```

Figura 7.21 ¿Qué hace este programa?

a) El número del vendedor.
b) El número del producto.
c) El valor total en dólares de ese producto vendido en ese día.

Así, cada vendedor pasa entre 0 y 5 notas de venta por día. Suponga que está disponible la información sobre todas las notas del mes pasado. Escriba un applet que lea toda esta información para las ventas del último mes y que resuma las ventas totales por vendedor, por producto. Todos los totales deben guardarse en el arreglo bidimensional **ventas**. Después de procesar toda la información del mes pasado, muestre los resultados en formato tabular, en donde cada columna represente a un vendedor específico y cada fila represente a un producto. Saque el total de cada fila para obtener las ventas totales de cada producto durante el último mes; saque el total de cada columna para obtener las ventas totales de cada vendedor durante el último mes. Su impresión tabular debe incluir estos totales cruzados a la derecha de las filas totalizadas, y en la parte inferior de las columnas totalizadas. Muestre los resultados en un objeto **JTextArea**.

7.21 (*Gráficos de tortuga.*) El lenguaje Logo hizo famoso el concepto de los *gráficos de tortuga*. Imagine a una tortuga mecánica que camina por todo el cuarto, bajo el control de un programa en Java. La tortuga sostiene una pluma en una de dos posiciones, arriba o abajo. Mientras la pluma está abajo, la tortuga va trazando figuras a medida que se va moviendo; mientras la pluma está arriba, la tortuga se mueve alrededor libremente, sin trazar nada. En este problema usted simulará la operación de la tortuga y creará un bloc de dibujo computarizado.

Utilice un arreglo de 20 por 20 llamado **piso**, que se inicialice con ceros. Lea los comandos de un arreglo que los contenga. Lleve el registro de la posición actual de la tortuga en todo momento, y si la pluma se encuentra arriba o abajo. Suponga que la tortuga siempre empieza en la posición (0, 0) del piso, con su pluma hacia arriba. El conjunto de comandos de la tortuga que su programa debe procesar se muestra en la figura 7.23.

Suponga que la tortuga se encuentra en algún lado cerca del centro del piso. El siguiente "programa" dibuja e imprime un cuadrado de 12 por 12, dejando la pluma en posición levantada:

```
2
5,12
```



```
3
5,12
3
5,12
3
5,12
1
6
9
```

A medida que la tortuga se vaya desplazando con la pluma hacia abajo, asigne **1** a los elementos apropiados del arreglo **piso**. Cuando se dé el comando **6** (imprimir), siempre que haya un **1** en el arreglo muestre un asterisco o cualquier carácter que usted elija. Siempre que haya un **0**, muestre un carácter en blanco.

Escriba un applet en java para implementar las herramientas de gráficos de tortuga aquí descritas. El applet debe mostrar los gráficos de tortuga en un objeto **JTextArea**, usando fuente Monoespaciada. Escriba varios programas de gráficos de tortuga para dibujar figuras interesantes. Agregue otros comandos para incrementar el poder de su lenguaje de gráficos de tortuga.

7.22 (*Paseo del caballo.*) Uno de los enigmas más interesantes para los entusiastas del ajedrez es el problema del Paseo del caballo, propuesto originalmente por el matemático Euler. ¿Puede la pieza de ajedrez, conocida como caballo, moverse alrededor de un tablero de ajedrez vacío y tocar cada una de las 64 posiciones una y sólo una vez? A continuación estudiaremos detalladamente este intrigante problema.

El caballero realiza solamente movimientos en forma de L (dos espacios en una dirección y un espacio en una dirección perpendicular). Por lo tanto, como se muestra en la figura 7.24, desde una posición cerca del centro de un tablero de ajedrez vacío, el caballo (etiquetado como C) puede hacer ocho movimientos distintos (numerados del 0 al 7).

a) Dibuje un tablero de ajedrez de ocho por ocho en una hoja de papel, e intente realizar un Paseo del caballo en forma manual. Ponga un **1** en la posición inicial, un **2** en la segunda posición, un **3** en la tercera, etc. Antes de empezar el paseo, estime qué tan lejos podrá avanzar, recordando que un paseo completo consta de 64 movimientos. ¿Qué tan lejos llegó? ¿Estuvo esto cerca de su estimación?

```
1  // Exercise 7.19: QueHaceEsto2.java
2  import java.awt.*;
3  import javax.swing.*;
4
5  public class QueHaceEsto2 extends JApplet {
6
7     public void init()
8     {
9        int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10       JTextArea areaSalida = new JTextArea();
11
12       unaFuncion( arreglo, 0, areaSalida );
13
14       Container contenedor = getContentPane();
15       contenedor.add( areaSalida );
16    }
17
18    public void unaFuncion( int arreglo2[], int x, JTextArea salida )
19    {
20       if ( x < arreglo2.length ) {
21          unaFuncion( arreglo2, x + 1, salida );
22          salida.append( arreglo2[ x ] + "  " );
23       }
24    }
25 }
```

Figura 7.22 ¿Qué hace este programa?

http://librosysolucionarios.net

Comando	Significado
1	Pluma arriba.
2	Pluma abajo.
3	Voltear a la derecha.
4	Voltear a la izquierda.
5, 10	Avanzar hacia delante 10 espacios (reemplace el 10 por un número distinto de espacios).
6	Imprimir el arreglo de 20 por 20.
9	Fin de los datos (centinela).

Figura 7.23 Comandos de gráficos de tortuga.

b) Ahora desarrollaremos un applet para mover el caballo alrededor de un tablero de ajedrez. El tablero estará representado por un arreglo bidimensional llamado `tablero`, de ocho por ocho. Cada posición se inicializará con cero. Describiremos cada uno de los ocho posibles movimientos en términos de sus componentes horizontales y verticales. Por ejemplo, un movimiento de tipo 0, como se muestra en la figura 7.24, consiste en mover dos posiciones horizontalmente a la derecha y una posición verticalmente hacia arriba. Un movimiento de tipo 2 consiste en mover una posición horizontalmente a la izquierda y dos posiciones verticalmente hacia arriba. Los movimientos horizontal a la izquierda y vertical hacia arriba se indican con números negativos. Los ocho movimientos pueden describirse mediante dos arreglos unidimensionales llamados `horizontal` y `vertical`, de la siguiente manera:

```
horizontal[ 0 ] = 2          vertical[ 0 ] = -1
horizontal[ 1 ] = 1          vertical[ 1 ] = -2
horizontal[ 2 ] = -1         vertical[ 2 ] = -2
horizontal[ 3 ] = -2         vertical[ 3 ] = -1
horizontal[ 4 ] = -2         vertical[ 4 ] = 1
horizontal[ 5 ] = -1         vertical[ 5 ] = 2
horizontal[ 6 ] = 1          vertical[ 6 ] = 2
horizontal[ 7 ] = 2          vertical[ 7 ] = 1
```

Deje que las variables `filaActual` y `columnaActual` indiquen la fila y columna, respectivamente, de la posición actual del caballo. Para hacer un movimiento de tipo `numeroMovimiento`, en donde `numeroMovimiento` puede estar entre 0 y 7, su programa debe utilizar las instrucciones

```
filaActual += vertical[ numeroMovimiento ];
columnaActual += horizontal[ numeroMovimiento];
```

Escriba un programa para mover el caballo alrededor del tablero de ajedrez. Utilice un contador que varíe de `1` a `64`. Registre la última cuenta en cada posición a la que se mueva el caballo. Evalúe cada movimiento potencial para ver si el caballo ya visitó esa posición. Pruebe cada movimiento potencial para asegurarse que el caballo no se salga del tablero de ajedrez. Ejecute el programa. ¿Cuántos movimientos hizo el caballo?

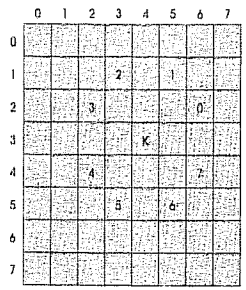


Figura 7.24 Los ocho posibles movimientos del caballo.

c) Después de intentar escribir y ejecutar un programa de Paseo del caballo, probablemente haya desarrollado algunas ideas valiosas. Utilizaremos estas ideas para desarrollar una *heurística* (o "regla empírica") para mover el caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora considerablemente la probabilidad de tener éxito. Probablemente usted ya observó que las posiciones externas son más difíciles que las posiciones cercanas al centro del tablero. De hecho, las posiciones más difíciles o inaccesibles son las cuatro esquinas.

La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que sea más fácil llegar, de manera que cuando el tablero se congestione cerca del final del paseo, habrá una mayor probabilidad de éxito.

Podríamos desarrollar una "heurística de accesibilidad" clasificando cada una de las posiciones de acuerdo a qué tan accesibles son y luego mover siempre el caballo (usando los movimientos en L del caballo) a la posición más inaccesible. Etiquetaremos un arreglo bidimensional llamado `accesibilidad` con números que indiquen desde cuántas posiciones es accesible una posición determinada. En un tablero de ajedrez en blanco, cada una de las 16 posiciones más cercanas al centro se clasifican con `8`; cada posición en la esquina se clasifica con `2`; y las demás posiciones tienen números de accesibilidad `3`, `4` o `6`, de la siguiente manera:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Escriba una nueva versión del Paseo del caballo, utilizando la heurística de accesibilidad. El caballo deberá moverse siempre a la posición con el número de accesibilidad más bajo. En caso de un empate, el caballo podrá moverse a cualquiera de las posiciones empatadas. Por lo tanto, el paseo puede empezar en cualquiera de las cuatro esquinas. [*Nota:* Al ir moviendo el caballo alrededor del tablero, su programa deberá reducir los números de accesibilidad a medida que se vayan ocupando más posiciones. De esta manera y en cualquier momento dado durante el paseo, el número de accesibilidad de cada una de las posiciones disponibles seguirá siendo igual al número preciso de posiciones desde las que se puede llegar a esa posición.] Ejecute esta versión de su programa. ¿Logró completar el paseo? Modifique el programa para realizar 64 paseos, en donde cada uno empiece desde una posición distinta en el tablero. ¿Cuántos paseos completos logró realizar?

d) Escriba una versión del programa del Paseo del caballo que, al encontrarse con un empate entre dos o más posiciones, decida qué posición elegir buscando más adelante aquellas posiciones que se puedan alcanzar desde las posiciones "empatadas". Su programa debe mover el caballo a la posición empatada para la cual el siguiente movimiento lo lleve a una posición con el número de accesibilidad más bajo.

7.23 (*Paseo del caballo: métodos de fuerza bruta.*) En la parte (c) del ejercicio 7.22, desarrollamos una solución al problema del Paseo del caballo. El método utilizado, llamado "*heurística de accesibilidad*", genera muchas soluciones y se ejecuta con eficiencia.

A medida que se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y algoritmos relativamente menos sofisticados. A éste le podemos llamar el método de la "fuerza bruta" para resolver problemas.

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (mediante sus movimientos legítimos en L) en forma aleatoria. Su programa debe ejecutar un paseo e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?

b) La mayoría de las veces, el programa de la parte (a) produce un paseo relativamente corto. Ahora modifique su programa para intentar 1000 paseos. Use un arreglo unidimensional para llevar el registro del número de paseos de cada longitud. Cuando su programa termine de intentar los 1000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

c) Es muy probable que el programa de la parte (b) le haya brindado algunos paseos "respetables", pero no completos. Ahora deje que su programa se ejecute hasta que produzca un paseo completo. (*Precaución:* Esta versión del programa podría ejecutarse durante horas en una computadora poderosa.) Una vez más, mantenga una tabla del número de paseos de cada longitud e imprímala cuando se encuentre el primer paseo completo. ¿Cuántos paseos intentó su programa antes de producir uno completo? ¿Cuánto tiempo se tomó?

http://librosysolucionarios.net

d) Compare la versión de la fuerza bruta del Paseo del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál requirió más poder de cómputo? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la heurística de accesibilidad? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la fuerza bruta? Argumente las ventajas y desventajas de solucionar el problema mediante la fuerza bruta en general.

7.24 (*Ocho reinas.*) Otro enigma para los entusiastas del ajedrez es el problema de las Ocho reinas, el cual pregunta lo siguiente: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna reina "ataque" a cualquier otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Use la idea desarrollada en el ejercicio 7.22 para formular una heurística para resolver el problema de las Ocho reinas. Ejecute su programa. (*Sugerencia:* Es posible asignar un valor a cada una de las posiciones en el tablero de ajedrez, para indicar cuántas posiciones de un tablero vacío se "eliminan" si una reina se coloca en esa posición. A cada una de las esquinas se le asignaría el valor 22, como se demuestra en la figura 7.25. Una vez que estos "números de eliminación" se coloquen en las 64 posiciones, una heurística apropiada podría ser la siguiente: coloque la siguiente reina en la posición con el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?

7.25 (*Ocho reinas: métodos de fuerza bruta.*) En este ejercicio usted desarrollará varios métodos de fuerza bruta para resolver el problema de las ocho reinas que presentamos en el ejercicio 7.24.

a) Utilice la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 7.23, para resolver el problema de las ocho reinas.

b) Utilice una técnica exhaustiva (es decir, pruebe todas las combinaciones posibles de las ocho reinas en el tablero) para resolver el problema de las ocho reinas.

c) ¿Por qué el método de la fuerza bruta exhaustiva podría no ser apropiado para resolver el problema del paseo del caballo?

d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva.

7.26 (*Paseo del caballo: prueba del paseo cerrado.*) En el paseo del caballo (ejercicio 7.22), se lleva a cabo un paseo completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un paseo cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de la posición en la que el caballo empezó el paseo. Modifique el programa que escribió en el ejercicio 7.22 para probar si el paseo ha sido completo, y si se trató de un paseo cerrado.

7.27 (*La criba de Eratóstenes.*) Un número primo es cualquier entero divisible sólo por sí mismo y por el número 1. La Criba de Eratóstenes es un método para encontrar números primos, el cual opera de la siguiente manera:

a) Crea un arreglo del tipo primitivo `boolean`, con todos los elementos inicializados en `true` (verdadero). Los elementos del arreglo con índices primos permanecerán como `true`. Cualquier otro elemento del arreglo eventualmente cambiará a `false`.

b) Empezando con el índice 2 del arreglo, determine si un elemento dado es `true`. De ser así, itere a través del resto del arreglo y asigne `false` a todo elemento cuyo índice sea múltiplo del índice del elemento que tiene el valor `true`. Después continúe el proceso con el siguiente elemento que tenga el valor `true`. Para el índice 2 del arreglo, todos los elementos más allá del elemento 2 en el arreglo que tengan índices múltiplos de 2 (los índices 4, 6, 8, 10, etc.) se establecerán en `false`; para el índice 3 del arreglo, todos los elementos más allá del elemento 3 en el arreglo que tengan índices múltiplos de 3 (los índices 6, 9, 12, 15, etc.) se establecerán en `false`; y así sucesivamente.

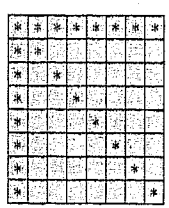


Figura 7.25 Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

Cuando este proceso termine, los elementos del arreglo que aún sean `true` indicarán que el índice es un número primo. Estos índices pueden mostrarse. Escriba un programa que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 2 y 999. Ignore los elementos 0 y 1 del arreglo.

7.28 (*Ordenamiento por cubetas.*) Un ordenamiento por cubetas empieza con un arreglo unidimensional de enteros positivos a ordenar, y un arreglo bidimensional de enteros con filas cuyos índices se encuentran entre el 0 y el 9, y columnas con índices entre el 0 y $n - 1$, en donde n es el número de valores a ordenar. A cada fila del arreglo bidimensional se le conoce como una cubeta. Escriba un applet que contenga un método llamado `ordenamCubetas`, el cual tome un arreglo entero como argumento y se ejecute de la siguiente manera:

a) Que coloque cada valor del arreglo unidimensional en una fila del arreglo en cubetas, con base en el valor del dígito de los "unos". Por ejemplo, 97 se coloca en la fila 7, 3 se coloca en la fila 3 y 100 se coloca en la fila 0. A este procedimiento se le conoce como un "paso de distribución".

b) Que itere a través del arreglo en cubetas fila por fila, y copie los valores de vuelta al arreglo original. A este procedimiento se le conoce como un "paso de recolección". El nuevo orden de los valores anteriores en el arreglo unidimensional es 100, 3 y 97.

c) Que repita este proceso para cada posición de los dígitos subsiguientes (decenas, cientos, miles, etcétera).

En la segunda pasada (dígito de decenas), el 100 se coloca en la fila 0, el 3 en la fila 0 (porque 3 no tiene dígito de decenas) y el 97 se coloca en la fila 9. Después del paso de recolección, el orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada (dígito de centenas), el 100 se coloca en la fila 1, el 3 se coloca en la fila 0 y el 97 en la fila 0 (después del 3). Después de este último paso de recolección, el arreglo original queda ordenado.

Observe que el arreglo bidimensional de cubetas tiene 10 veces el tamaño del arreglo entero que se va a ordenar. Esta técnica de ordenamiento ofrece un mejor rendimiento que el ordenamiento burbuja, pero requiere de más memoria; el método burbuja requiere espacio solamente para un elemento adicional de datos. Esta comparación es un ejemplo del intercambio espacio-tiempo: el ordenamiento en cubetas utiliza más memoria que el ordenamiento burbuja, pero tiene un mejor desempeño. En esta versión del ordenamiento burbuja se requiere copiar todos los datos de vuelta al arreglo original en cada pasada. Otra posibilidad es crear un segundo arreglo bidimensional en cubetas e intercambiar repetidamente los datos entre los dos arreglos en cubetas.

7.29 (*Simulación: La tortuga y la liebre.*) En este problema usted recreará la clásica carrera de la tortuga y la liebre. Utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores empezarán la carrera en la "posición 1" de 70 posiciones. Cada posición representa a una posible posición a lo largo del curso de la carrera. La línea de meta se encuentra en la posición 70. El primer competidor en llegar a la posición 70 recibirá una cubeta llena con zanahorias y lechuga frescas. El recorrido se abre paso hasta la cima de una resbalosa montaña, por lo que ocasionalmente los competidores pierden terreno.

Un reloj hace tictac una vez por segundo. Con cada tic del reloj, su applet debe ajustar la posición de los animales de acuerdo con las reglas de la figura 7.26. Use variables para llevar el registro de las posiciones de los animales (los números son del 1 al 70). Empiece con cada animal en la posición 1 (la "puerta de inicio"). Si un animal se resbala hacia la izquierda antes de la posición 1, regréselo a la posición 1.

Genere los porcentajes en la figura 7.26 produciendo un entero aleatorio i en el rango $1 \leq i \leq 10$. Para la tortuga, realice un "paso pesado rápido" cuando $1 \leq i \leq 5$, un "resbalón" cuando $6 \leq i \leq 7$ o un "paso pesado lento" cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento actual
Tortuga	Paso pesado rápido	50%	3 posiciones a la derecha
	Resbalón	20%	6 posiciones a la izquierda
	Paso pesado lento	30%	1 posición a la derecha
Liebre	Dormir	20%	Ningún movimiento
	Gran salto	20%	9 posiciones a la derecha
	Gran resbalón	10%	12 posiciones a la izquierda
	Pequeño salto	30%	1 posición a la derecha
	Pequeño resbalón	20%	2 posiciones a la izquierda

Figura 7.26 Reglas para ajustar las posiciones de la tortuga y la liebre.

http://librosysolucionarios.net

Empiece la carrera imprimiendo el mensaje

```
¡¡¡PUM!!!
¡¡¡Y ARRANCAN!!!
```

Luego, para cada tic del reloj (es decir, cada repetición de un ciclo) imprima una línea de 70 posiciones, mostrando la letra `T` en la posición de la tortuga y la letra `H` en la posición de la liebre. En ocasiones los competidores se encontrarán en la misma posición. En este caso, la tortuga muerde a la liebre y su programa debe imprimir `¡¡¡OUCH!!!` Empezando en esa posición, Todas las posiciones de impresión distintas de la `T`, la `H` o el mensaje `¡¡¡OUCH!!!` (en caso de un empate) deben estar en blanco.

Después de imprimir cada línea, compruebe si uno de los animales ha llegado o se ha pasado de la posición 70. De ser así, imprima quién fue el ganador y termine la simulación. Si la tortuga gana, imprima `¡¡¡LA TORTUGA GANA!!!` `¡¡¡YAY!!!` Si la liebre gana, imprima `La liebre gana. Qué mal.` Si ambos animales ganan en el mismo tic del reloj, tal vez usted quiera favorecer a la tortuga (la más débil) o tal vez quiera imprimir `Es un empate.` Si ninguno de los dos animales gana, ejecute el ciclo de nuevo para simular el siguiente tic del reloj. Cuando esté listo para ejecutar su programa, reúna a un grupo de aficionados para que vean la carrera. ¡Se sorprenderá al ver lo participativa que puede ser su audiencia!

Posteriormente presentaremos una variedad de herramientas de Java, como gráficos, imágenes, animación, sonido y subprocesamiento múltiple. Cuando estudie esas herramientas, tal vez pueda disfrutar mejorando su simulación de la tortuga y la liebre.

7.30 La serie de Fibonacci

0, 1, 1, 2, 3, 4, 8, 13, 21, ...

empieza con los términos 0 y 1, y tiene la propiedad de que cada término sucesivo es la suma de los dos términos anteriores.

a) Escriba un método *no recursivo* llamado `fibonacci( n )` que calcule el *enésimo* número de Fibonacci. Incorpore este método en un applet que permita al usuario introducir el valor de *n*.
b) Determine el número de Fibonacci más grande que puede imprimirse en su sistema.
c) Modifique el programa que escribió en la parte (a), de manera que utilice `double` en vez de `int` para calcular y devolver números de Fibonacci, y utilice este programa modificado para repetir la parte (b).

EJERCICIOS DE RECURSIVIDAD

7.31 (*Ordenamiento por selección.*) Un ordenamiento por selección busca en un arreglo el elemento más pequeño, e intercambia ese elemento con el primer elemento del arreglo. El proceso se repite para el subarreglo que comienza con el segundo elemento. Cada pasada del arreglo coloca a un elemento en su posición correcta. Para un arreglo de *n* elementos se requieren *n* – 1 pasadas, y para cada subarreglo se deben realizar *n* – 1 comparaciones para encontrar el valor más pequeño. Cuando el subarreglo que se está procesando contiene un elemento, el arreglo se ordena. Escriba el método recursivo `OrdenamSeleccion` para realizar este algoritmo.

7.32 (*Palíndromos.*) Un palíndromo es una cadena que se deletrea igual al derecho y al revés. Algunos ejemplos de palíndromos son "radar", "ojo", "oso". Escriba un método recursivo llamado `pruebaPalindromo` que devuelva el valor `true` tipo `boolean` si la cadena guardada en el arreglo es un palíndromo, y `false` en caso contrario. El método debe ignorar espacios y puntuación en la cadena. [*Sugerencia:* Utilice el método `toCharArray` de la clase `String`, el cual no toma argumentos, para obtener un arreglo `char` que contenga los caracteres en el objeto `String`. Luego pase el arreglo al método `pruebaPalindromo`.]

7.33 (*Búsqueda lineal.*) Modifique el programa de la figura 7.11 para utilizar el método recursivo `busquedaLineal` para realizar una búsqueda lineal en el arreglo. Este método debe recibir como argumentos un arreglo entero, la longitud de ese arreglo y la clave de búsqueda. Si se encuentra la clave de búsqueda, devuelva su índice en el arreglo; en caso contrario devuelva –1.

7.34 (*Búsqueda binaria.*) Modifique la figura 7.12 para utilizar el método recursivo `busquedaBinaria` para realizar una búsqueda binaria en el arreglo. Este método debe recibir como argumentos un arreglo entero, el índice inicial y el índice final. Si se encuentra la clave de búsqueda, devuelva su índice en el arreglo; en caso contrario, devuelva –1.

7.35 (*Ocho reinas.*) Modifique el programa de las ocho reinas que creó en el ejercicio 7.24 para resolver el problema en forma recursiva.

7.36 (*Imprimir un arreglo.*) Escriba un método recursivo llamado `imprimirArreglo` que tome un arreglo de valores `int` y la longitud del arreglo como argumentos, y que no devuelva nada. Este método deberá dejar de procesarse y regresar cuando reciba un arreglo de longitud cero.

7.37 (*Imprimir un arreglo al revés.*) Escriba un método recursivo llamado `cadenaInversa` que tome un arreglo de caracteres que contenga una cadena como argumento, imprima la cadena al revés y no devuelva nada. [*Sugerencia:* Use el método `toCharArray` de `String`, el cual no toma argumentos, para obtener un arreglo `char` que contenga los caracteres en el objeto `String`. Después, pase el arreglo al método `pruebaPalindromo`.]

7.38 (*Encontrar el valor mínimo en un arreglo.*) Escriba un método recursivo llamado `minimoRecursivo` que tome como argumentos a un arreglo entero y su longitud, y que devuelva el elemento más pequeño del arreglo. Este método debe dejar de procesarse y regresar cuando reciba un arreglo de un elemento.

7.39 (*Quicksort.*) La técnica de ordenamiento recursivo llamada *Quicksort* utiliza el siguiente algoritmo básico para un arreglo unidimensional de valores:

a) *Paso de particionamiento*: Toma el primer elemento del arreglo desordenado y determina su ubicación final en el arreglo ordenado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que ese elemento, y todos los valores a la derecha del elemento en el arreglo son mayores que ese elemento). Ahora tenemos un elemento en su posición correcta y dos subarreglos desordenados.
b) *Paso recursivo*: Realiza el paso 1 en cada subarreglo desordenado.

Cada vez que se Realiza el paso 1 en un subarreglo, se coloca otro elemento en su posición final en el arreglo ordenado, y se crean dos subarreglos desordenados. Cuando un subarreglo consiste de un elemento, ese elemento se encuentra en su posición final (pues un arreglo de un elemento se encuentra ya ordenado).

El algoritmo básico parece bastante simple, pero ¿cómo determinamos la posición final del primer elemento de cada subarreglo? Como ejemplo, considere el siguiente conjunto de valores (el elemento en negritas es el de particionamiento; se colocará en su ubicación final en el arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

a) Empezando desde la derecha, compare cada elemento con **37** hasta que encuentre uno menor que **37**; e intercámbielos. El primer elemento menor que **37** es 12, por lo que se intercambian **37** y 12. El nuevo arreglo es

12 2 6 4 89 8 10 **37** 68 45

El elemento 12 se encuentra en cursiva para indicar que se acaba de intercambiar con **37**.

b) Empezando desde la parte izquierda con el primer elemento después de 12, compare cada elemento con 37 hasta que encuentre uno mayor que **37** e intercámbielos. El primer elemento mayor que **37** es 89, por lo que se intercambian el **37** y 89. El nuevo arreglo es

12 2 6 4 **37** 8 10 89 68 45

c) Empezando desde la derecha, con el primer elemento antes de 89, compare cada elemento con **37** hasta que se encuentre uno menor que **37** e intercámbielos. El primer elemento menor que **37** es 10, por lo que se intercambian **37** y 10. El nuevo arreglo es

12 2 6 4 *10* 8 **37** 89 68 45

d) Empezando desde la izquierda, con el primer elemento después de 10, compare cada elemento con **37** hasta que se encuentre uno mayor que **37** e intercámbielos. Ya no hay elementos mayores que **37**, por lo que sabemos que **37** se ha colocado en su posición final en el arreglo ordenado.

Una vez que se ha aplicado la partición en el arreglo anterior, hay dos subarreglos desordenados. El subarreglo con valores menores que 37 contiene los elementos 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene los elementos 89, 68 y 45. El ordenamiento continúa particionando ambos subarreglos de la misma forma que el arreglo original.

Con base en la discusión anterior, escriba un método recursivo llamado `quickSort` para ordenar un arreglo entero unidimensional. Este método debe recibir como argumentos un arreglo entero, un índice inicial y un índice final. El método `particion` debe ser llamado por `quickSort` para realizar el paso de particionamiento.

7.40 (*Recorrido de laberintos.*) La siguiente malla de gatos (#) y puntos (.) es un arreglo bidimensional que representa a un laberinto. Los gatos representan las paredes del laberinto, y los puntos representan los pasillos en las posibles rutas a través del laberinto. Sólo pueden hacerse movimientos hacia una ubicación en el arreglo que contenga un punto.

http://librosysolucionarios.net

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Hay un algoritmo simple para recorrer un laberinto que garantiza encontrar la salida (suponiendo que haya una). Si no hay una salida, llegará de nuevo a la ubicación inicial. Coloque su mano derecha en la pared a su derecha y empiece a caminar hacia delante. Nunca quite su mano derecha de la pared. Si el laberinto da vuelta a la derecha, siga la pared hacia la derecha. Mientras usted no quite su mano de la pared, tarde o temprano llegará a la salida del laberinto. Tal vez haya una ruta más corta que la que usted ha tomado, pero está garantizado que usted saldrá del laberinto si sigue el algoritmo.

Escriba un método recursivo llamado `recorridoLaberinto` para recorrer el laberinto. Este método debe recibir como argumentos un arreglo de caracteres de 12 por 12 como representación del laberinto, y la ubicación inicial del mismo. Mientras `recorridoLaberinto` trata de localizar la salida, debe colocar el carácter `X` en cada posición de la ruta. El método debe mostrar el laberinto después de cada movimiento, para que el usuario pueda observar cómo se resuelve el laberinto.

7.41 (*Generación aleatoria de laberintos.*) Escriba un método llamado `generadorLaberintos` que tome como argumento un arreglo de caracteres bidimensional de 12 por 12, y que produzca un laberinto en forma aleatoria. El método debe también proporcionar las ubicaciones inicial y final del laberinto. Pruebe su método `recorridoLaberinto` del ejercicio 7.40, utilizando varios laberintos generados al azar.

7.42 (*Laberintos de cualquier tamaño.*) Generalice los métodos `recorridoLaberinto` y `generadorLaberintos` de los ejercicios 7.40 y 7.41, para procesar laberintos de cualquier anchura y altura.

SECCIÓN ESPECIAL: CONSTRUYA SU PROPIA COMPUTADORA

En los siguientes problemas nos desviaremos temporalmente del mundo de la programación en lenguajes de alto nivel, para "abrir de par en par" una computadora y ver su estructura interna. Presentaremos la programación en lenguaje máquina y escribiremos varios programas en este lenguaje. Para que ésta sea una experiencia valiosa, ¡crearemos también una computadora (mediante la técnica de la simulación basada en software) en la que pueda ejecutar sus programas en lenguaje máquina!

7.43 (*Programación en lenguaje máquina.*) Vamos a crear una computadora a la que llamaremos Simpletron. Como su nombre lo implica, es una máquina simple, pero poderosa. Simpletron sólo ejecuta programas escritos en el único lenguaje que entiende directamente: el lenguaje máquina de Simpletron , o LMS.

Simpletron contiene un acumulador, un "registro especial" en el cual se coloca la información antes de que Simpletron la utilice en los cálculos, o que la analice de distintas maneras. Toda la información dentro de Simpletron se manipula en términos de palabras. Una palabra es un número decimal con signo de cuatro dígitos, tal como +3364, –1293, +0007 y –0001. Simpletron está equipada con una memoria de 100 palabras, y se hace referencia a estas palabras mediante sus números de ubicación 00, 01, ..., 99.

Antes de ejecutar un programa LMS debemos cargar, o colocar, el programa en memoria. La primera instrucción de cada programa LMS se coloca siempre en la ubicación 00. El simulador empezará a ejecutarse en esta ubicación.

Cada instrucción escrita en LMS ocupa una palabra de la memoria de Simpletron (y por lo tanto, las instrucciones son números decimales de cuatro dígitos con signo). Vamos a suponer que el signo de una instrucción LMS siempre será positivo, pero el signo de una palabra de información puede ser positivo o negativo. Cada una de las ubicaciones en la memoria de Simpletron puede contener una instrucción, un valor de datos utilizado por un programa o un área no utilizada (y por lo tanto indefinida) de memoria. Los primeros dos dígitos de cada instrucción LMS son el código de operación que especifica la operación a realizar. Los códigos de operación de LMS se sintetizan en la figura 7.27.

Los últimos dos dígitos de una instrucción LMS son el operando (la dirección de la ubicación en memoria que contiene la palabra a la cual se aplica la operación). Consideremos varios programas simples en LMS.

El primer programa en LMS (figura 7.28) lee dos números del teclado, calcula e imprime su suma. La instrucción `+1007` lee el primer número del teclado y lo coloca en la ubicación `07` (que se ha inicializado con 0). Después, la instrucción `+1008` lee el siguiente número y lo coloca en la ubicación `08`. La instrucción carga, `+2007`, coloca el primer número en el acumulador y la instrucción suma, `+3008`, suma el segundo número al número en el acumulador. Todas las instrucciones LMS aritméticas dejan sus resultados en el acumulador. La instrucción almacena, `+2109`, coloca el resultado de vuelta en la ubicación de memoria `09`, desde la cual la instrucción escribe, `+1109`, toma el número y lo imprime (como un número decimal de cuatro dígitos con signo). La instrucción alto, `+4300`, termina la ejecución.

El segundo programa en LMS (figura 7.29) lee dos números desde el teclado, determina e imprime el valor más grande. Observe el uso de la instrucción `+4107` como una transferencia de control condicional, en forma muy similar a la instrucción `if` de Java.

Código de operación	Significado
Operaciones de entrada/salida:	
`final int LEE = 10;`	Lee una palabra desde el teclado y la introduce en una ubicación específica de memoria.
`final int ESCRIBE = 11;`	Escribe una palabra de una ubicación específica de memoria y la imprime en la pantalla.
Operaciones de carga/almacenamiento:	
`final int CARGA = 20;`	Carga una palabra de una ubicación específica de memoria y la coloca en el acumulador.
`final int ALMACENA = 21;`	Almacena una palabra del acumulador dentro de una ubicación específica de memoria.
Operaciones aritméticas:	
`final int SUMA = 30;`	Suma una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int RESTA = 31;`	Resta una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int DIVIDE = 32;`	Divide una palabra de una ubicación específica de memoria entre la palabra en el acumulador (deja el resultado en el acumulador).
`final int MULTIPLICA = 33;`	Multiplica una palabra de una ubicación específica de memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`final int BIFURCA = 40;`	Bifurca hacia una ubicación específica de memoria.
`final int BIFURCANEG = 41;`	Bifurca hacia una ubicación específica de memoria si el acumulador es negativo.
`final int BIFURCACERO = 42;`	Bifurca hacia una ubicación específica de memoria si el acumulador es cero.
`const int ALTO = 43;`	Alto. El programa completó su tarea.

Figura 7.27 Códigos de operación del Lenguaje máquina Simpletron (LMS).

Ubicación	Número	Instrucción
00	`+1007`	(Lee A)
01	`+1008`	(Lee B)
02	`+2007`	(Carga A)

Figura 7.28 Programa en LMS que lee dos enteros y calcula la suma. (Parte 1 de 2.)

http://librosysolucionarios.net

Ubicación	Número	Instrucción
03	+3008	(Suma B)
04	+2109	(Almacena C)
05	+1109	(Escribe C)
06	+4300	(Alto)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Resultado C)

Figura 7.28 Programa en LMS que lee dos enteros y calcula la suma. (Parte 2 de 2.)

Ahora escriba programas en LMS para realizar cada una de las siguientes tareas:

a) Usar un ciclo controlado por centinela para leer 10 números positivos. Calcular e imprimir la suma.
b) Usar un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcular e imprimir su promedio.
c) Leer una serie de números, determinar e imprimir el número más grande. El primer número leído indica cuántos números deben procesarse.

7.44 (*Un simulador de computadora.*) En este problema usted va a crear su propia computadora. No, no va a soldar componentes, sino que utilizará la poderosa técnica de la simulación basada en software para crear un modelo de software orientado a objetos de Simpletron, la computadora del ejercicio 7.43. Su simulador Simpletron convertirá la computadora que usted utiliza en Simpletron, y será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 7.43. Su Simpletron será un applet controlado por eventos: usted hará clic en un botón para ejecutar cada instrucción LMS, y podrá ver a la instrucción "en acción".

Cuando ejecute su simulador Simpletron, debe empezar mostrando lo siguiente:

```
*** ¡Bienvenido a Simpletron!  ***
*** Por favor, introduzca en su programa una instrucción   ***
*** (o palabra de datos) a la vez en el campo de texto de  ***
*** entrada. Yo le mostraré el número de ubicación y       ***
*** un signo de interrogación (?). Entonces usted          ***
*** escribirá la palabra para esa ubicación. Oprima el     ***
*** botón Terminar para dejar de introducir su programa.   ***
```

El programa debe mostrar un objeto `JTextField` llamado entrada, en el que el usuario escribirá cada instrucción, una a la vez, y un botón `Terminar` para que el usuario haga clic en él cuando haya introducido el programa completo en LMS. Simule la memoria del Simpletron con un arreglo unidimensional llamado `memoria`, que cuente con 100 elementos. Ahora suponga que el simulador se está ejecutando y examinaremos el diálogo a medida que introduzcamos el programa de la figura 7.29 (ejercicio 7.43):

Ubicación	Número	Instrucción
00	+1009	(Lee A)
01	+1010	(Lee B)
02	+2009	(Carga A)
03	+3110	(Resta B)
04	+4107	(Bifurcación negativa a 07)
05	+1109	(Escribe A)
06	+4300	(Alto)
07	+1110	(Escribe B)
08	+4300	(Alto)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 7.29 Programa en LMS que lee dos enteros y determina cuál de ellos es mayor.

http://librosysolucionarios.net


```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
```

Su programa debe usar un objeto `JTextField` para mostrar la ubicación en memoria, seguida por un signo de interrogación. Cada uno de los valores a la derecha de un signo de interrogación es introducido por el usuario en el objeto `JTextField` llamado entrada. Al hacer clic en el botón `Terminar`, el programa debe mostrar lo siguiente:

```
*** Se completó la carga del programa  ***
*** Empieza la ejecución del programa  ***
```

Ahora el programa en LMS se ha colocado (o cargado) en el arreglo `memoria`. Simpletron debe proporcionar un botón llamado `"Ejecutar siguiente instrucción"`, en el cual el usuario pueda hacer clic para ejecutar cada instrucción del programa en LMS. La ejecución comienza con la instrucción en la ubicación `00` y, como en Java, continúa secuencialmente a menos que se lleve a otra parte del programa mediante una transferencia de control.

Use la variable `acumulador` para representar el registro acumulador. Use la variable `contadorDeInstrucciones` para llevar el registro de la ubicación en memoria que contiene la instrucción que se está ejecutando. Use la variable `codigoDeOperacion` para indicar la operación que se esté realizando actualmente (es decir, los dos dígitos a la izquierda en la palabra de instrucción). Use la variable `operando` para indicar la ubicación de memoria en la que va a operar la instrucción actual. Por lo tanto, `operando` está compuesta por los dos dígitos más a la derecha de la instrucción que se esté ejecutando en esos momentos. No ejecute las instrucciones directamente desde la memoria. En vez de eso, transfiera la siguiente instrucción a ejecutar desde la memoria hasta una variable llamada `registroDeInstruccion`. Luego "recoja" los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después "recoja" los dos dígitos a la derecha y colóquelos en `operando`. Cada uno de los registros anteriores debe tener su correspondiente objeto `JTextField` en el que se muestre su valor actual, en todo momento. Cuando Simpletron comience con la ejecución, todos los registros especiales se deben inicializar con cero.

Ahora vamos a "dar un paseo" por la ejecución de la primera instrucción LMS, `+1009` en la ubicación de memoria `00`. A este procedimiento se le conoce como ciclo de ejecución de una instrucción.

El `contadorDeInstrucciones` nos indica la ubicación de la siguiente instrucción a ejecutar. Nosotros buscamos el contenido de esa ubicación de memoria, utilizando la siguiente instrucción de Java:

```
registroDeInstruccion = memoria[ contadorDeInstrucciones ];
```

El código de operación y el operando se extraen del registro de instrucción, mediante las instrucciones

```
codigoDeOperacion = registroDeInstruccion / 100;
operando = registroDeInstruccion % 100;
```

Ahora, Simpletron debe determinar que el código de operación es en realidad un *lee* (en comparación con un *escribe*, *carga*, etcétera). Una instrucción `switch` establece la diferencia entre las 12 operaciones de LMS. En la instrucción `switch` se simula el comportamiento de varias instrucciones LMS, como se muestra en la figura 7.30. En breve hablaremos sobre las instrucciones de bifurcación y dejaremos las otras a usted.

Cuando el programa en LMS termine de ejecutarse, deberán mostrarse el nombre y contenido de cada registro, así como el contenido completo de la memoria. A este tipo de impresión se le denomina *vaciado de la computadora*. Para ayudarlo a programar su método de vaciado, en la figura 7.31 se muestra un formato de vaciado de muestra. Observe que un vaciado, después de la ejecución de un programa de Simpletron, muestra los valores actuales

Instrucción	Descripción
lee:	Mostrar un cuadro de diálogo de entrada con el indicador "`Escriba un entero.`" Convertir el valor introducido en un entero y almacenarlo en la ubicación `memoria[ operando ]`.
carga:	`acumulador = memoria[ operando ];`
suma:	`acumulador += memoria[ operando ];`

Figura 7.30 Comportamiento de algunas instrucciones SML en el simpletron.

de las instrucciones y los valores de los datos al momento en que se terminó la ejecución. El vaciado de muestra supone que la salida se enviará a la pantalla mediante una serie de llamadas a los métodos `System.out.print` y `System.out.println`. Sin embargo, le recomendamos que experimente con una versión que pueda mostrarse en el subprograma utilizando un objeto `JTextArea`, o un arreglo de objetos `JTextField`.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa, `+1009` en la ubicación `00`. Como lo hemos indicado, la instrucción `switch` simula esta tarea pidiendo al usuario que escriba un valor en el cuadro de diálogo de entrada, leyendo el valor, convirtiéndolo en entero y almacenándolo en la ubicación de memoria `memoria[ operando ]`. Ya que su Simpletron es controlada por eventos, espera a que el usuario escriba un valor en el objeto `JTextField` llamado entrada y oprima *Intro*. A continuación, el valor se lee y se coloca en la ubicación `09`.

En este punto se ha completado la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Como la instrucción que acaba de ejecutarse no es una transferencia de control, sólo necesitamos incrementar el registro contador de instrucción de la siguiente manera:

```
++contadorDeInstrucciones;
```

Esta acción completa la ejecución simulada de la primera instrucción. Cuando el usuario hace clic en el botón `Ejecutar siguiente instrucción`, todo el proceso (es decir, el ciclo de ejecución de una instrucción) empieza de nuevo, con la búsqueda de la siguiente instrucción a ser ejecutada.

Ahora veremos cómo se simulan las instrucciones de bifurcación (las transferencias de control). Todo lo que necesitamos hacer es ajustar el valor en el contador de instrucciones de manera apropiada. Por lo tanto, la instrucción de bifurcación condicional (`40`) se simula dentro de la instrucción `switch` como

```
contadorDeInstrucciones = operando;
```

```
REGISTROS:
acumulador                +0000
contadorDeInstrucciones     00
registroDeInstruccion     +0000
codigoDeOperacion           00
operando                    00

MEMORIA:
         0     1     2     3     4     5     6     7     8     9
  0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Figura 7.31 Un vaciado de muestras.

http://librosysolucionarios.net

La instrucción condicional "bifurcar si el acumulador es cero" se simula como

```
if ( acumulador == 0 )
   contadorDeInstrucciones = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar cada uno de los programas que escribió en el ejercicio 7.43. Si lo desea, puede embellecer al LMS con características adicionales y ofrecerlas en su simulador.

Su simulador debe comprobar diversos tipos de errores. Por ejemplo, durante la fase de carga del programa, cada número que el usuario escribe en la `memoria` de Simpletron debe encontrarse dentro del rango de `-9999` a `+9999`. Su simulador debe probar que cada número introducido se encuentre dentro de este rango y, en caso contrario, seguir pidiendo al usuario que vuelva a introducir el número hasta que introduzca un número correcto.

Durante la fase de ejecución, su simulador debe comprobar varios errores graves, como los intentos de dividir entre cero, intentos de ejecutar códigos de operación inválidos, y desbordamientos del acumulador (es decir, las operaciones aritméticas que den como resultado valores mayores que `+9999` o menores que `-9999`). Dichos errores graves se conocen como *errores fatales*. Al detectar un error fatal, su simulador deberá imprimir un mensaje de error tal como:

```
*** Intento de dividir entre cero ***
*** La ejecución de Simpletron se terminó en forma anormal ***
```

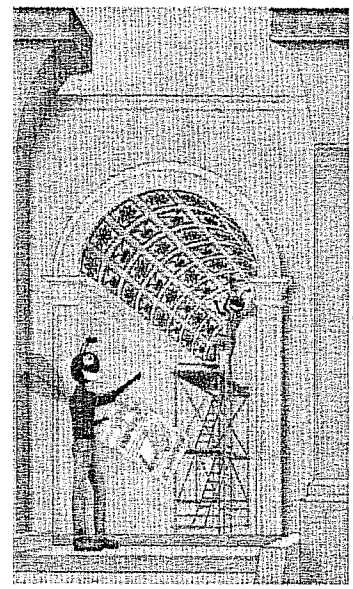
y deberá imprimir un vaciado de computadora completo en el formato que vimos anteriormente. Este análisis ayudará al usuario a localizar el error en el programa.

7.45 (*Modificaciones al simulador Simpletron.*) En el ejercicio 7.44 usted escribió una simulación de software de una computadora que ejecuta programas escritos en el Lenguaje máquina Simpletron (LMS). En este ejercicio proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 20.26 y 20.27 propondremos la creación de un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel (una variación de Basic) a Lenguaje máquina Simpletron. Algunas de las siguientes modificaciones y mejoras pueden requerirse para ejecutar los programas producidos por el compilador:

a) Extienda la memoria del simulador Simpletron, de manera que contenga 1000 ubicaciones de memoria para permitir a Simpletron manejar programas más grandes.
b) Permita al simulador realizar cálculos de residuo. Esta modificación requiere de una instrucción adicional en lenguaje máquina Simpletron.
c) Permita al simulador realizar cálculos de exponenciación. Esta modificación requiere una instrucción adicional en lenguaje máquina Simpletron.
d) Modifique el simulador para que pueda utilizar valores hexadecimales, en vez de valores enteros para representar instrucciones en lenguaje máquina Simpletron.
e) Modifique el simulador para permitir la impresión de una nueva línea. Esta modificación requiere una instrucción adicional en lenguaje máquina Simpletron.
f) Modifique el simulador para procesar valores de punto flotante además de valores enteros.
g) Modifique el simulador para manejar la introducción de cadenas. (*Sugerencia:* Cada palabra de Simpletron puede dividirse en dos grupos, cada una de las cuales guarda un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal de código ASCII de un carácter. Agregue una instrucción de lenguaje máquina que reciba como entrada una cadena y la almacene, empezando en una ubicación de memoria específica de Simpletron. La primera mitad de la palabra en esa ubicación será una cuenta del número de caracteres en la cadena [es decir, la longitud de la cadena]. Cada media palabra subsiguiente contiene un carácter ASCII, expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y lo asigna a una media palabra.)
h) Modifique el simulador para manejar la impresión de cadenas almacenadas en el formato de la parte (g). (*Sugerencia:* Agregue una instrucción en lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria de Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena [es decir, la longitud de la misma]. Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente.)

8

Programación basada en objetos



Objetivos

- Comprender el concepto de encapsulamiento y ocultamiento de datos.
- Comprender las nociones de la abstracción de datos y los tipos de datos abstractos (ADTs).
- Crear ADTs en Java (clases).
- Crear y utilizar objetos.
- Controlar el acceso a las variables de instancia y los métodos.
- Comprender el uso de la referencia `this`.
- Utilizar las variables y métodos de las clases.
- Apreciar el valor de la orientación a objetos.

Mis objetivos, todos sublimes
deberé conseguirlos a tiempo.
W. S. Gilbert

¿Es éste un mundo en el que se pueden ocultar las virtudes?
William Shakespeare

Sus servidores públicos le sirven bien.
Adlai Stevenson

Pero qué, para servir a nuestros fines privados,
¿Se olvida el engaño a nuestros amigos?
Charles Churchill

Esto por encima de todo: que tu propio ser sea verdadero.
William Shakespeare

No tengas amigos que no sean iguales a ti.
Confucio

http://librosysolucionarios.net

Plan general

8.1 Introducción
8.2 Implementación de un tipo de dato abstracto con una clase
8.3 Alcance de las clases
8.4 Control del acceso a los miembros
8.5 Referencias a los miembros del objeto actual mediante `this`
8.6 Inicialización de los objetos de una clase: Constructores
8.7 Uso de constructores sobrecargados
8.8 Uso de los métodos *set (establecer)* y *get (obtener)*
8.9 Composición
8.10 Recolección de basura
8.11 Miembros de clase estáticos
8.12 Variables de instancia finales
8.13 Creación de paquetes
8.14 Acceso a los paquetes
8.15 Reutilización de software
8.16 Abstracción de datos y encapsulamiento
8.17 (Ejemplo práctico opcional) Acerca de los objetos: Cómo empezar a programar las clases para la simulación del elevador

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

8.1 Introducción

En este capítulo investigaremos acerca de la orientación a objetos en Java. Algunos lectores podrían preguntar, ¿por qué aplazamos este tema hasta ahora? Existen varias razones. En primer lugar, los objetos que crearemos en este capítulo están parcialmente compuestos de piezas de programación estructurada. Para explicar la organización de los objetos, necesitábamos establecer una base en la programación estructurada con las estructuras de control. También queríamos estudiar los métodos detalladamente, antes de introducir la orientación a objetos. Por último, queríamos que usted se familiarizara con los arreglos, que son objetos de Java.

En nuestras discusiones sobre los programas orientados a objetos en capítulos anteriores, presentamos muchos conceptos básicos (por ejemplo, "pensar en objetos") y terminología (como, "hablar de objetos") que se relacionan con la programación orientada a objetos (POO) en Java. También hablamos sobre nuestra metodología para desarrollar programas: analizamos problemas típicos que requerían la creación de applet o aplicaciones y determinamos cuáles clases de la API de Java eran necesarias para implementar cada programa. Luego seleccionamos las variables y los métodos apropiados para cada programa, y especificamos la manera en que un objeto de nuestra clase debería colaborar con los objetos de las clases de la API de Java, para lograr los objetivos del programa en general.

Ahora repasaremos brevemente algunos conceptos y terminología clave de la orientación a objetos. Esta tecnología utiliza clases para *encapsular* (es decir, envolver) datos (atributos) y métodos (comportamientos). Por ejemplo, el estéreo de un auto encapsula todos los atributos y comportamientos que le permiten al conductor del auto seleccionar una estación de radio, o reproducir cintas o CDs. Las compañías que fabrican autos no fabrican los estéreos, sino que los compran y simplemente los conectan en el tablero de cada auto. Los componentes del radio están encapsulados en su caja.

El encapsulamiento permite a los objetos ocultar su implementación de otros objetos; a este principio se le conoce como *ocultamiento de la información*. Aunque los objetos pueden comunicarse entre sí, a través de

interfaces bien definidas (de la misma forma que la interfaz de un conductor para un auto incluye un volante, pedal del acelerador, pedal del freno y palanca de velocidades), no están conscientes de cómo se implementan otros objetos (de la misma forma que el conductor no conoce cómo están implementados los mecanismos del volante, motor, freno y transmisión). Por lo general, los detalles de implementación se ocultan dentro de los mismos objetos. Evidentemente es posible conducir un auto en forma efectiva sin conocer los detalles sobre el funcionamiento de los motores, transmisiones y sistemas de escape. De manera similar, es posible seleccionar una estación de radio en un estéreo de auto sin necesidad de saber cómo funciona el radio. Más adelante veremos por qué el ocultamiento de la información es tan imprescindible para la buena ingeniería de software.

En los *lenguajes de programación por procedimientos* (como C), la programación tiende a ser *orientada a la acción*. Sin embargo, la programación en Java está *orientada a objetos*. En los lenguajes de programación por procedimientos, la unidad de programación es la *función* (en Java las funciones se llaman *métodos*). En Java, la unidad de programación es la *clase*. Los objetos eventualmente se *instancian* (es decir, se crean) a partir de estas clases, y los atributos y comportamientos están encapsulados dentro de los "límites" de las clases, en forma de campos y métodos.

Los programadores, al crear programas por procedimientos, se concentran en la escritura de funciones. Agrupan acciones que realizan cierta tarea en una función y luego agrupan varias funciones para formar un programa. Evidentemente los datos son importantes en los programas por procedimientos, pero existen principalmente para apoyar las acciones que realizan las funciones. Los *verbos* en un documento de requerimientos del sistema que describen los requerimientos para una nueva aplicación, ayudan a un programador, que crea programas por procedimientos, a determinar el conjunto de funciones que trabajarán entre sí para implementar el sistema.

En contraste, los programadores en Java se concentran en la creación de sus propios tipos de referencia, mejor conocidos como *clases*. Cada clase contiene como sus *miembros* a un conjunto de campos (variables) y métodos que manipulan a esos campos. (En otros lenguajes de programación, a los campos de una clase se les conoce como *miembros de datos*.[1]) Los *sustantivos* en un documento de requerimientos del sistema ayudan al programador en Java a determinar un conjunto inicial de clases con las que se va a empezar el proceso de diseño. Los programadores utilizan esas clases para instanciar objetos que trabajen en conjunto para implementar el sistema.

En este capítulo explicaremos cómo crear y utilizar clases y objetos, un tema conocido como *programación basada en objetos (PBO)*. En los capítulos 9 y 10 presentaremos la *herencia* y el *polimorfismo*, respectivamente; tecnologías clave que permiten la programación orientada a objetos. Aunque no hablaremos sobre la herencia detalladamente sino hasta el capítulo 9, ésta forma parte de varias declaraciones de clases en este capítulo, y se ha utilizado en varios ejemplos anteriormente. Por ejemplo, todas las clases de applet definidas hasta este punto heredaron de la clase `JApplet`.

8.2 Implementación de un tipo de dato abstracto con una clase

Las clases en Java facilitan la creación de *ADT's (tipos de datos abstractos)*, los cuales ocultan su implementación a los clientes (o usuarios de la clase). Un problema en los lenguajes de programación por procedimientos es que el código cliente a menudo depende de los detalles de implementación de los datos utilizados en el código. Esta dependencia tal vez requiera de reescribir el código cliente, si la implementación de los datos cambia. Los ADTs eliminan este problema al proporcionar *interfaces* (conjuntos de métodos ofrecidos por las clases) independientes de la implementación a sus clientes. Es posible para el creador de una clase cambiar la implementación interna de esa clase sin afectar a los clientes de esa clase.

Observación de ingeniería de software 8.1



Es importante escribir programas que sean comprensibles y fáciles de mantener. El cambio es la regla, en vez de ser la excepción. Los programadores deben anticipar que su código será modificado. Como veremos, las clases facilitan la capacidad de modificación de los programas.

1. Algunas veces utilizamos tecnología informal que sea conocida para los programadores en otros lenguajes, como "miembros de datos", en vez de la terminología específica de Java, como "campos". Para obtener un listado de la terminología específica de Java consulte la *Especificación del lenguaje Java*, la cual puede descargarse del sitio Web `java.sun.com/docs/books/jls/index.html`.

http://librosysolucionarios.net

Declaración de la clase Tiempo1

El siguiente ejemplo consiste de dos clases: **Tiempo1** (figura 8.1) y **PruebaTiempo1** (figura 8.2). La clase **Tiempo1** (declarada en el archivo **Tiempo1.java**) se utiliza para crear objetos que representen la hora. La clase **PruebaTiempo1** (declarada en un archivo separado, llamado **PruebaTiempo1.java**) es una clase de aplicación, en la cual el método **main** crea un objeto de la clase **Tiempo1** e invoca a sus métodos. Estas clases *deben* declararse en archivos separados, ya que ambas son clases **public**. De hecho, cada declaración de clase que comience con la palabra clave **public** debe guardarse en un archivo que tenga exactamente el mismo nombre que la clase, y que termine con la extensión **.java** en su nombre de archivo. [*Nota:* Los resultados de este programa aparecen en la figura 8.2.]



Error común de programación 8.1

*Declarar más de una clase **public** en el mismo archivo es un error de sintaxis.*

```
1  // Fig. 8.1: Tiempo1.java
2  // Declaración de la clase Tiempo1 que mantiene la hora en formato de 24 horas.
3  import java.text.DecimalFormat;
4
5  public class Tiempo1 extends Object {
6     private int hora;      // 0 - 23
7     private int minuto;    // 0 - 59
8     private int segundo;   // 0 - 59
9
10    // El constructor de Tiempo1 inicializa cada variable de instancia en cero;
11    // se asegura de que cada objeto Tiempo1 inicie en un estado consistente
12    public Tiempo1()
13    {
14       establecerHora( 0, 0, 0 );
15    }
16
17    // establecer un nuevo valor de tiempo utilizando hora universal; realizar
18    // comprobaciones de validez en los datos; establecer valores inválidos en
         cero
19    public void establecerHora( int h, int m, int s )
20    {
21       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
22       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
23       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
24    }
25
26    // convertir a String en formato de hora universal
27    public String aStringUniversal()
28    {
29       DecimalFormat dosDigitos = new DecimalFormat( "00" );
30
31       return dosDigitos.format( hora ) + ":" +
32          dosDigitos.format( minuto ) + ":" + dosDigitos.format( segundo );
33    }
34
35    // convertir a String en formato de hora estándar
36    public String aStringEstandar()
37    {
38       DecimalFormat dosDigitos = new DecimalFormat( "00" );
39
40       return ( (hora == 12 || hora == 0) ? 12 : hora % 12 ) + ":" +
```

Figura 8.1 Implementación del tipo de datos abstracto **Tiempo1** como una clase. (Parte 1 de 2.)

http://librosysolucionarios.net


```
41          dosDigitos.format( minuto ) + ":" + dosDigitos.format( segundo ) +
42          ( hora < 12 ? " AM" : " PM" );
43    }
44
45 } // fin de la clase Tiempo1
```

Figura 8.1 Implementación del tipo de datos abstracto **Tiempo1** como una clase. (Parte 2 de 2.)

En la línea 5 de la figura 8.1 comienza la declaración de la clase **Tiempo1**. En esta declaración se indica que **Tiempo1** extiende a (mediante **extends**) la clase ***Object*** (del paquete **java.lang**). Los programadores en Java utilizan la *herencia* para crear clases a partir de clases existentes. De hecho, todas las clases en Java (excepto **Object**) *extienden a* (heredan de) una clase existente. En la línea 5, la palabra clave **extends** seguida por el nombre de la clase **Object** indica que la clase **Tiempo1** hereda los atributos y comportamientos existentes de la clase **Object**. Si la declaración de una clase no especifica **extends** y el nombre de una clase a la derecha del nuevo nombre de la clase, esta nueva clase extiende implícitamente a la clase **Object**. De hecho, cada clase de Java contiene los 11 métodos declarados en la clase **Object**. No es necesario comprender la herencia para aprender los conceptos y programas en este capítulo. Exploraremos la herencia y la clase **Object** detalladamente en el capítulo 9.

El *cuerpo* de la declaración de la clase está delimitado por las llaves izquierda y derecha ({ y }) en las líneas 5 y 45. Cualquier información que coloquemos en el cuerpo se dice que está encapsulada (es decir, envuelta) en la clase. Por ejemplo, la clase **Tiempo1** contiene tres valores enteros: **hora**, **minuto** y **segundo**, los cuales representan la hora en formato de *hora universal* (formato de *reloj de 24 horas*). Anteriormente nos referimos a las variables declaradas en la declaración de una clase, pero no dentro de la declaración de un método, como campos. A un campo que no se declara como **static** se le conoce como *variable de instancia*; cada instancia (objeto) de la clase contiene su propia copia separada de las variables de instancia de la clase. (Hablaremos sobre los campos **static** en la sección 8.11.)

Las palabras clave ***public*** y ***private*** son *modificadores de acceso*. Las variables o métodos declarados como **public** pueden utilizarse en cualquier parte del programa en que haya una referencia a un objeto de la clase. Las variables o métodos declarados con el modificador de acceso **private** pueden ser utilizadas *solamente* por los métodos de la clase en la que están declarados.



Buena práctica de programación 8.1

*Se debe anteponer un modificador de acceso a cualquier declaración de un campo o método. Como regla empírica, los campos deben declararse como **private** y los métodos como **public**.*

Las tres variables de instancia enteras **hora**, **minuto** y **segundo** se declaran (líneas 6 a 8) con el modificador de acceso **private**, lo cual indica que estas variables de instancia son accesibles solamente para los métodos de la clase; esto se conoce como *ocultamiento de los datos*. Cuando un programa crea (instancia) un objeto de la clase **Tiempo1**, dichas variables se encapsulan en el objeto y pueden utilizarse sólo mediante métodos de la clase de ese objeto (comúnmente a través de los métodos **public** de la clase). Por lo general, los campos se declaran como **private** y los métodos como **public**. Es posible tener métodos **private** y campos **public**, como veremos más adelante. Los métodos **private** se conocen como *métodos utilitarios* o *métodos ayudantes*, ya que pueden ser llamados sólo por otros métodos de esa clase, y se utilizan para dar soporte a la operación de esos métodos. El uso de campos **public** es poco común, además de ser una práctica de programación peligrosa.



Buena práctica de programación 8.2

*Preferimos enlistar primero los campos **private** de una clase para que, a medida que usted lea el código, pueda ver los nombres y tipos de las variables antes de que sean utilizados en los métodos de la clase.*



Observación de ingeniería de software 8.2

*Haga que el miembro de una clase sea **private** si no hay razón para que se utilice fuera de la declaración de la clase.*

A menudo las clases incluyen métodos de acceso que pueden leer o devolver datos. Otro uso común para los métodos de acceso es probar si una condición es verdadera o falsa; dichos métodos se llaman comúnmente *métodos predicado*. Un ejemplo de un método predicado sería el método `estaVacio` para una *clase contenedora*: una clase capaz de almacenar muchos objetos, como una lista enlazada, una pila o una cola. (Describiremos detalladamente a las estructuras de datos en los capítulos 20, 21 y 22.) Un programa podría probar `estaVacio` antes de intentar leer otro elemento de un objeto contenedor. Un programa podría probar `estaLleno` antes de intentar insertar otro elemento en un objeto contenedor.

La clase `Tiempo1` contiene el constructor `Tiempo1` (líneas 12 a 15) y los métodos `establecerHora` (líneas 19 a 24), `aStringUniversal` (líneas 27 a 33) y `aStringEstandar` (líneas 36 a 43). Éstos son los métodos `public` (también conocidos como *servicios `public`* o *interfaces `public`*) de la clase. Los *clientes* de la clase `Tiempo1` como la clase `PruebaTiempo1` (figura 8.2), utilizan los métodos `public` de `Tiempo1` para manipular los datos almacenados en objetos `Tiempo1` o para hacer que la clase `Tiempo1` realice algún servicio. Los clientes de una clase utilizan referencias para interactuar con un objeto de la clase. Por ejemplo, el método `paint` de un applet es un cliente de la clase `Graphics`. El método `paint` utiliza su argumento (una referencia a un objeto `Graphics`, como `g`) para dibujar en el applet, llamando a los métodos que son servicios `public` de la clase `Graphics` (como `drawString`, `drawLine`, `drawOval` y `drawRect`). Como veremos en la figura 8.2, se utilizará una referencia a `Tiempo1` para interactuar con un objeto `Tiempo1`.

En las líneas 12 a 15 se declara el *constructor* de la clase `Tiempo1`. Un constructor inicializa los objetos de una clase. Cuando un programa crea un objeto de la clase `Tiempo1`, la instrucción `new` asigna memoria para el objeto y llama al constructor para que inicialice a ese objeto. El constructor de la clase `Tiempo1` llama al método `establecerHora` (líneas 19 a 24) para inicializar las variables de instancia `hora`, `minuto` y `segundo` con `0` (representando la medianoche). Los constructores pueden tomar argumentos, pero no pueden devolver valores. Una diferencia importante entre los constructores y los métodos es que los constructores *no pueden especificar un tipo de valor de retorno* (ni siquiera `void`). El nombre del constructor debe ser el mismo que el de la clase. Por lo general, los constructores se declaran como `public`.



Error común de programación 8.2

Tratar de declarar un tipo de valor de retorno para un constructor, o tratar de devolver (mediante `return`) un valor en un constructor, es un error. Java permite que otros métodos de la clase tengan el mismo nombre de ésta y que especifiquen tipos de valor de retorno. Dichos métodos no son constructores y no serán llamados al instanciar a un objeto de la clase. Java determina qué métodos son constructores localizando los métodos que tienen el mismo nombre que la clase y que no especifican un tipo de valor de retorno.

El método `establecerHora` (líneas 19 a 24) es un método `public` que declara tres parámetros `int` y los utiliza para ajustar la hora. Una expresión condicional evalúa cada argumento para determinar si el valor se encuentra en un rango especificado. Por ejemplo, el valor de `hora` debe ser mayor o igual a 0 y menor que 24 (línea 21), ya que el formato de hora universal representa a las horas como enteros del 0 al 23. De manera similar, los valores de los minutos y segundos deben ser mayores o iguales a 0 y menores que 60 (líneas 22 y 23). Cualquier valor fuera de estos rangos es un valor inválido y se establece en cero de manera predeterminada, para asegurar que un objeto `Tiempo1` siempre contenga datos válidos. (En este ejemplo, cero es un valor válido para `hora`, `minuto` y `segundo`.) Esto también se conoce como *mantener el objeto en un estado consistente* o *mantener la integridad del objeto*. Cuando `establecerHora` recibe valores de datos inválidos, es conveniente que el programa indique que la hora es inválida. En el capítulo 15 veremos el manejo de excepciones, que puede utilizarse para indicar valores de inicialización inválidos.



Buena práctica de programación 8.3

Incluya constructores en la declaración de una clase, para asegurar que las variables de instancia de la clase contengan valores válidos cuando se creen objetos de esa clase.

El método `aStringUniversal` (líneas 27 a 33) no toma argumentos y devuelve una cadena en formato de hora universal, el cual consiste de seis dígitos: dos para la hora, dos para el minuto y dos para el segundo. Por ejemplo, si la hora fuera 1:30:07 PM, el método `aStringUniversal` devolvería `13:30:07`. En la línea 29 se crea una instancia de la clase `DecimalFormat` (importada en la línea 3 del paquete `java.text`) para dar formato a la hora universal. A la referencia `dosDigitos` se le asigna una referencia a un objeto de

la clase `DecimalFormat`, la cual se inicializa con el patrón `"00"`. Esto indica que un número con formato debe consistir de dos dígitos; cada `0` representa la posición de un dígito. Si el número al que se está dando formato es un solo dígito, se le agrega automáticamente un `0` a la izquierda (es decir, `8` se formatea como `08`). La instrucción `return` de las líneas 31 y 32 utiliza a `dosDigitos` para invocar al método `format` de `DecimalFormat` (el cual devuelve una cadena que contiene el número formateado) para dar formato a los valores de `hora`, `minuto` y `segundo` como cadenas de dos dígitos. Esas cadenas se concatenan con el operador `+` (separadas por signos de dos puntos) y se devuelve la cadena resultante mediante el método `aString Universal`.

El método `aStringEstandar` (líneas 36 a 43) no toma argumentos y devuelve un valor `String` en formato de hora estándar, el cual consiste de los valores `hora`, `minuto` y `segundo` separados por signos de dos puntos y seguidos de un indicador AM o PM (por ejemplo, `1:27:06 PM`). Al igual que el método `aStringUniversal`, el método `aStringEstandar` utiliza el método `format` de `DecimalFormat` para dar formato a `minuto` y `segundo` como valores de dos dígitos, con ceros a la izquierda en caso de ser necesario. En la línea 40 se determina el valor para `hora` en la cadena; si la `hora` es `0` o `12` (`AM` o `PM`), este valor aparece como `12`; en caso contrario, `hora` aparece como un valor entre 1 y 11.

Cómo utilizar la clase `Tiempo1`

Después de declarar la clase, podemos utilizarla como un tipo en las declaraciones como:

```
Tiempo1 puestaDeSol;    // referencia a un objeto Tiempo1
```

El nombre de la clase (`Tiempo1`) es un nombre de tipo. Una clase puede producir muchos objetos, de igual forma que un tipo primitivo, como `int`, puede producir muchas variables. Los programadores pueden declarar nuevos tipos de clases conforme lo necesiten; ésta es una de las razones por las que Java se conoce como un *lenguaje extensible*.

La clase de aplicación `PruebaTiempo1` de la figura 8.2 utiliza a la clase `Tiempo1`. El método `main` de la clase `PruebaTiempo1` declara y crea una instancia de la clase `Tiempo1` llamada `tiempo`, en la línea 9. Al instanciarse el objeto, `new` asigna la memoria en la que el objeto `Tiempo1` se almacenará, después llama al constructor de `Tiempo1` para inicializar las variables de instancia del nuevo objeto `Tiempo1`. El constructor invoca al método `establecerHora` para inicializar cada variable de instancia explícitamente con `0`. Después `new` regresa una referencia al nuevo objeto y esa referencia se asigna a `tiempo`. De manera similar, en las líneas 29 y 38 de la clase `Tiempo1` (figura 8.1) se utiliza `new` para crear objetos `DecimalFormat` y pasar el argumento `"00"` al constructor de cada objeto para indicar el patrón utilizado para dar formato.



Observación de ingeniería de software 8.3

Cada vez que `new` crea un objeto de una clase, se hace una llamada al constructor de esa clase para inicializar las variables de instancia del nuevo objeto.

```
1  // Fig. 8.2: PruebaTiempo1.java
2  // Clase PruebaTiempo1 que utiliza la clase Tiempo1.
3  import javax.swing.JOptionPane;
4
5  public class PruebaTiempo1 {
6
7     public static void main( String args[] )
8     {
9        Tiempo1 tiempo = new Tiempo1(); // llamar al constructor de Tiempo1
10
11       // anexar versión String de tiempo a salida String
12       String salida = "La hora universal inicial es: " +
13          tiempo.aStringUniversal() + "\nLa hora estándar inicial es: " +
14          tiempo.aStringEstandar();
15
```

Figura 8.2 Un objeto `Tiempo1` utilizado en un programa. (Parte 1 de 2.)

http://librosysolucionarios.net

```
16        // cambiar tiempo y anexar hora actualizada a salida
17        tiempo.establecerHora( 13, 27, 6 );
18        salida += "\n\nLa hora universal después de establecerHora es: " +
19           tiempo.aStringUniversal() +
20           "\nLa hora estándar después de establecerHora es:
21              " + tiempo.aStringEstandar();
22
23        // establecer tiempo con valores inválidos; anexar tiempo actualizado
24           a salida
25        tiempo.establecerHora( 99, 99, 99 );
26        salida += "\n\nDespués de intentar ajustes inválidos: " +
27           "\nHora universal: " + tiempo.aStringUniversal() +
28           "\nHora estándar: " + tiempo.aStringEstandar();
29
30        JOptionPane.showMessageDialog( null, salida,
31           "Prueba de la clase Tiempo1", JOptionPane.INFORMATION_MESSAGE );
32
33        System.exit( 0 );
34
35     } // fin de main
36
37  } // fin de la clase PruebaTiempo1
```

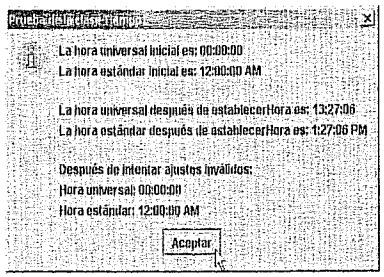


Figura 8.2 Un objeto `Tiempo1` utilizado en un programa. (Parte 2 de 2.)

Observe que el archivo `PruebaTiempo1.java` no importa la clase `Tiempo1`. Si una clase se encuentra en el mismo paquete que la clase que la va a utilizar, la declaración `import` no se requiere. Toda clase en Java es parte de un paquete. Si un programador no especifica un paquete para una clase, ésta se coloca en un *paquete predeterminado* sin nombre, el cual incluye a todas las clases compiladas en el directorio actual, que no se hayan colocado explícitamente en un paquete. Debemos especificar declaraciones `import` para las clases de la API de Java, ya que están declaradas fuera del paquete de cada nueva clase que creamos. (En la sección 8.13 le enseñaremos cómo declarar sus propios paquetes.) Hay que tener en cuenta que las declaraciones `import` no se requieren si el programa califica completamente el nombre de cada clase, anteponiendo a éste el nombre de su paquete y un punto (`.`). Por ejemplo, un programa puede invocar al método `showMessageDialog` de la clase `JOptionPane` de la siguiente manera:

```
javax.swing.JOptionPane.showMessageDialog( "Aquí va su mensaje" );
```

Sin embargo, es bastante molesto usar estos nombres largos.

En las líneas 12 a 14 se declara la referencia `salida` tipo `String`, la cual representa una cadena que se mostrará en un cuadro de diálogo de mensaje. Inicialmente, el programa asigna a `salida` la hora en formato de hora universal (invocando al método `aStringUniversal` del objeto `tiempo`) y en formato de hora estándar (invocando al método `aStringEstandar` del objeto `tiempo`) para confirmar que el objeto `Tiempo1` se haya inicializado apropiadamente. Observe la sintaxis de la llamada al método en cada caso: la referencia

`tiempo` va seguida de un punto (`.`) y del nombre del método. La referencia se refiere al objeto que recibirá la llamada al método (es decir, el objeto al cual se enviará el mensaje).

En la línea 17 se utiliza el objeto `tiempo` para invocar al método `establecerHora` y cambiar la hora. Después, en las líneas 18 a 20 se anexa nuevamente la hora a `salida`, en ambos formatos para confirmar que la hora se haya ajustado correctamente.

Para ilustrar que el método `establecerHora` valida los valores que recibe, en la línea 23 se hace una llamada al método `establecerHora` y se intenta establecer las variables de instancia a valores inválidos de 99. Después, en las líneas 24 a 26 se anexa la hora a `salida` otra vez, en ambos formatos para confirmar que `establecerHora` haya validado los datos. En las líneas 28 y 29 se muestra un cuadro de diálogo de mensaje con los resultados de nuestro programa. Las últimas dos líneas de la ventana de salida muestran que el tiempo se restableció a medianoche (el valor predeterminado de un objeto `Tiempo1`) después de tratar de ajustar el tiempo con tres valores inválidos.

La clase `Tiempo1` es nuestro primer ejemplo de una clase que no es un applet, y que no contiene un método `main`. La clase `PruebaTiempo1` declara al método `main`, por lo que `PruebaTiempo1` puede utilizarse para iniciar la ejecución del programa. El método `main` se conoce como el *punto de entrada* en el programa. Una clase que contiene a `main` se conoce como una *clase de aplicación* o *clase ejecutable*.

Observaciones acerca de la declaración de la clase `Tiempo1`

Debemos tomar en cuenta algunas cuestiones relacionadas con el diseño de clases, con respecto a la clase `Tiempo1`. Las variables de instancia `hora`, `minuto` y `segundo` se declaran como `private`, por lo que no son directamente accesibles fuera de la clase en la que están declaradas. La representación actual de los datos utilizados dentro de la clase no es de la incumbencia de los clientes de la clase. Por ejemplo, sería razonable para la clase representar la hora internamente como el número de segundos transcurridos desde medianoche. Los clientes podrían utilizar los mismos métodos `public` y obtener los mismos resultados sin necesidad de saber esto.[2] En este sentido, se dice que la implementación de una clase está *oculta* de sus clientes.

Observación de ingeniería de software 8.4



El ocultamiento de la información fomenta la capacidad de modificar los programas y simplifica la percepción que tiene el cliente acerca de una clase.

En la figura 8.1, el constructor de `Tiempo1` inicializa las variables de instancia en 0 (es decir, el equivalente en hora universal de 12 AM). Esto asegura que el objeto se cree en un estado consistente (es decir, que todas las variables de instancia sean válidas). No pueden guardarse valores inválidos en las variables de instancia de un objeto `Tiempo1`, ya que se hace una llamada al constructor cuando se crea el objeto `Tiempo1`, y los intentos subsiguientes por parte de un cliente de modificar las variables de instancia, son examinados a fondo por el método `establecerHora`.

Las variables de instancia de una clase se inicializan en el constructor de esa clase, pero también pueden inicializarse al momento de declararse en el cuerpo de la clase. Si usted no inicializa las variables de instancia explícitamente, el compilador las inicializa implícitamente: las variables numéricas de tipo primitivo se establecen en cero, los valores `boolean` se establecen en `false` y las referencias se establecen en `null`.

Es interesante que los métodos `aStringUniversal` y `aStringEstandar` no tomen argumentos. Estos métodos pertenecen a la misma clase que las variables de instancia `hora`, `minuto` y `segundo`. Por lo tanto, tienen, en forma implícita, acceso a las variables de instancia del objeto `Tiempo1` específico en el que se invoquen. Esto hace que las llamadas a los métodos sean más concisas que las llamadas a funciones convencionales en la programación por procedimientos. También se reduce la probabilidad de pasar los argumentos erróneos, los tipos erróneos de argumentos o el número erróneo de los mismos, como ocurre comúnmente con las llamadas a funciones en el lenguaje C.

Observación de ingeniería de software 8.5



Utilizar una metodología de programación orientada a objetos a menudo simplifica las llamadas a los métodos, al reducirse el número de argumentos que deben pasarse. Este beneficio de la programación orientada a objetos se deriva del encapsulamiento de las variables de instancia y los métodos dentro de un objeto.

2. En el ejercicio 8.18 le pediremos que realice precisamente esta modificación a la clase `Tiempo1` de la figura 8.1, para mostrar que no hay un cambio visible para los clientes de la clase.

http://librosysolucionarios.net

Las clases simplifican la programación, ya que el cliente sólo necesita preocuparse de los métodos `public` expuestos por una clase. Por lo general, estos métodos están diseñados para ser orientados al cliente, en vez de ser orientados a la implementación. Los clientes no conocen nada de, ni están involucrados en, la implementación de una clase. Las interfaces cambian con menos frecuencia que las implementaciones. Cuando cambia una implementación, el código dependiente de la implementación debe cambiar de manera acorde. Al ocultar la implementación, reducimos la posibilidad de que otras partes del programa se vuelvan dependientes de los detalles de implementación de la clase.

8.3 Alcance de las clases

Recuerde que en la sección 6.9 vimos que las variables y los métodos de una clase pertenecen al *alcance de esa clase*. Dentro del alcance de una clase, los miembros de ésta son accesibles para todos los métodos de esa clase y pueden referenciarse directamente por su nombre.[3] Fuera del alcance de una clase, los miembros de ésta no pueden referenciarse por su nombre. Los miembros de una clase (como los miembros `public`) que sean visibles pueden utilizarse sólo a través de un "manejador": una referencia a un objeto de la clase, o el nombre mismo de la clase para los miembros `static`. Se puede hacer referencia a los miembros mediante *nombreReferenciaObjeto*`.`*nombreMiembroObjeto*. Por ejemplo, un programa puede determinar el número de elementos en el arreglo `calificaciones` accediendo al miembro `public` llamado `length` del arreglo, a través de la referencia `calificaciones`, como en `calificaciones.length`.

Las variables que se declaran en un método son conocidas sólo por ese método (es decir, son variables locales). Si un método declara a una variable local con el mismo nombre que una variable declarada en la clase que encierra a ese método, la variable declarada en el alcance de la clase queda sombreada (oculta) por la variable local. Una variable de instancia oculta puede utilizarse en el método si se antepone a su nombre la palabra clave `this` y un punto (`.`), como en `this.`*nombreDeVariable*. Describiremos la palabra clave `this` con más detalle en la sección 8.5.

8.4 Control del acceso a los miembros

Los modificadores de acceso `public` y `private` controlan el acceso a las variables y los métodos de una clase.[4] Como dijimos en la sección 8.2, el principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los *servicios* que la clase proporciona (la interfaz pública de la clase). Los clientes de la clase no necesitan preocuparse acerca de cómo la clase realiza sus tareas. Por esta razón, las variables y métodos `private` de una clase (es decir, los detalles de implementación de la clase) no son directamente accesibles para los clientes de una clase. Al proceso de restringir el acceso a los miembros de una clase mediante la palabra clave `private` se le conoce como ocultamiento de la información.

En la figura 8.3 se muestra que los miembros de clase `private` no son directamente accesibles fuera de la clase. En la línea 10 se trata de acceder directamente a la variable de instancia `private` llamada `hora`, del objeto `Tiempo1` que es referenciado por `tiempo`. Al compilar este programa, el compilador genera mensajes de error indicando que los miembros `private` `hora`, `minuto` y `segundo` no son accesibles. (*Nota:* Este programa supone que empleará la clase `Tiempo1` de la figura 8.1.)



Error común de programación 8.3

Si un método que no sea miembro de una clase específica, trata de acceder a uno de los miembros `private` de esa clase, se produce un error de sintaxis.

```
1  // Fig. 8.3: PruebaTiempo2.java
2  // Errores que se producen al tratar de acceder a los miembros private de
   Tiempo1.
```

Figura 8.3 Los miembros privados de la clase `Tiempo1` no son accesibles. (Parte 1 de 2.)

3. En la sección 8.11 veremos que los métodos `static` son una excepción a esta regla.
4. En el capítulo 9 presentaremos el modificador de acceso adicional `protected`.

http://librosysolucionarios.net


```
3  public class PruebaTiempo2 {
4
5     public static void main( String args[] )
6     {
7        Tiempo1 hora = new Tiempo1();
8
9        tiempo.hora = 7;    // error: hora es una variable de instancia privada
10       tiempo.minuto = 15; // error: minuto es una variable de instancia privada
11       tiempo.segundo = 30;// error: segundo es una variable de instancia privada
12    }
13
14 } // fin de la clase PruebaTiempo2
```

```
PruebaTiempo2.java:9: hora has private access in Tiempo1
      tiempo.hora = 7;    // error: hora es una variable de instancia privada
             ^
PruebaTiempo2.java:10: minuto has private access in Tiempo1
      tiempo.minuto = 15; // error: minuto es una variable de instancia privada
             ^
PruebaTiempo2.java:11: segundo has private access in Tiempo1
      tiempo.segundo = 30; // error: segundo es una variable de instancia privada
             ^
3 errors
```

Figura 8.3 Los miembros privados de la clase `Tiempo1` no son accesibles. (Parte 2 de 2.)

8.5 Referencias a los miembros del objeto actual mediante `this`

Todo objeto puede hacer referencia a sí mismo, mediante la palabra clave ***`this`*** (lo que se conoce como una *referencia* ***`this`***). En un método, la referencia `this` puede utilizarse en forma implícita y explícita para hacer referencia a las variables de instancia y otros métodos del objeto en el cual se haya llamado el método.

Ahora mostraremos el uso implícito y explícito de la referencia `this`, para permitir que el método `main` de la clase `PruebaThis` muestre los datos `private` de un objeto `HoraSimple` (figura 8.4). La clase `HoraSimple` (líneas 22 a 55) declara tres variables de instancia `private`: `hora`, `minuto` y `segundo` (líneas 23 a 25). El constructor (líneas 29 a 34) recibe tres argumentos `int` para inicializar un objeto `HoraSimple`. Observe que para este ejemplo hemos hecho que los nombres de los parámetros para el constructor (línea 29) sean idénticos a los nombres de las variables de instancia para la clase (líneas 23 a 25). Hicimos esto para mostrar el uso explícito de `this`. Si un método contiene una variable local con el mismo nombre que un campo de esa clase, ese método se referirá a la variable local, en vez de referirse al campo. En este caso, la variable local sombrea al campo en el alcance de ese método. Sin embargo, el método puede utilizar a `this` para hacer referencia al campo sombreado en forma explícita, como se muestra en las líneas 31 a 33 para las variables de instancia sombreadas de la clase `HoraSimple`.

```
1  // Fig. 8.4: PruebaThis.java
2  // Uso de la referencia this para hacer referencia a los campos y métodos.
3  import javax.swing.*;
4  import java.text.DecimalFormat;
5
6  public class PruebaThis {
7
8     public static void main( String args[] )
9     {
```

Figura 8.4 Uso de `this` en forma implícita y explícita para referirse a los miembros de un objeto. (Parte 1 de 2.)

```
10          HoraSimple hora = new HoraSimple( 12, 30, 19 );
11
12          JOptionPane.showMessageDialog( null, hora.crearCadena(),
13             "Demostración de la referencia \"this\"",
14             JOptionPane.INFORMATION_MESSAGE );
15
16          System.exit( 0 );
17       }
18
19    } // fin de la clase PruebaThis
20
21    // la clase HoraSimple muestra el uso de la referencia "this"
22    class HoraSimple {
23       private int hora;
24       private int minuto;
25       private int segundo;
26
27       // el constructor utiliza nombres de parámetros idénticos a los nombres
            de la variable
28       // de instancia; se requiere la referencia "this" para diferenciar entre
            los nombres
29       public HoraSimple( int hora, int minuto, int segundo )
30       {
31          this.hora = hora;          // establecer hora del objeto "this"
32          this.minuto = minuto;      // establecer minuto del objeto "this"
33          this.segundo = segundo;    // establecer segundo del objeto "this"
34       }
35
36       // usar "this" explícito e implícito para llamar al método aStringEstandar
37       public String crearCadena()
38       {
39          return "this.aStringEstandar(): " + this.aStringEstandar() +
40             "\naStringEstandar(): " + aStringEstandar();
41       }
42
43       // devolver la representación String de HoraSimple
44       public String aStringEstandar()
45       {
46          DecimalFormat dosDigitos = new DecimalFormat( "00" );
47
48          // "this" no se requiere aquí, ya que el método no tiene
49          // variables locales con los mismos nombres que las variables de
               instancia
50          return dosDigitos.format( this.hora ) + ":" +
51             dosDigitos.format( this.minuto ) + ":" +
52             dosDigitos.format( this.segundo );
53       }
54
55    } // fin de la clase HoraSimple
```

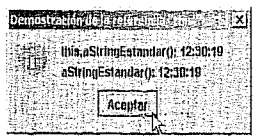


Figura 8.4 Uso de `this` en forma implícita y explícita para referirse a los miembros de un objeto. (Parte 2 de 2.)

http://librosysolucionarios.net

El método `crearCadena` (líneas 37 a 41) devuelve un `String` creado por una instrucción que utiliza la referencia `this` en forma explícita e implícita. En la línea 39 se utiliza a `this` explícitamente para llamar al método `aStringEstandar`. Sin embargo, en la línea 40 se utiliza a `this` en forma implícita para llamar al mismo método. Observe que ambas líneas realizan la misma tarea. Por lo tanto, los programadores generalmente no utilizan la referencia `this` en forma explícita para hacer referencia a los métodos dentro del objeto actual.

Error común de programación 8.4

Para un método en el cual un parámetro o variable local tenga el mismo nombre que un campo de la clase, debe usar la referencia `this` si desea tener acceso al campo de la clase; de no ser así, estará haciendo referencia al parámetro o variable local del método.

Tip para prevenir errores 8.1

Evite los nombres de parámetros o variables locales en los métodos que tengan conflicto con los nombres de los campos. Esto ayuda a prevenir errores sutiles, difíciles de localizar.

Buena práctica de programación 8.4

El uso explícito de la referencia `this` puede incrementar la claridad del programa en algunos contextos en donde `this` es opcional.

La clase `PruebaThis` (líneas 6 a 19) ejecuta la aplicación que muestra el uso explícito de `this`. En la línea 10 se crea una instancia de la clase `HoraSimple` e invoca a su constructor. En las líneas 12 a 14 se invoca el método `crearCadena` del objeto `HoraSimple` y después se muestran los resultados al usuario en un cuadro de diálogo de mensaje. Los ejemplos subsiguientes en este capítulo y en los capítulos posteriores mostrarán cómo puede utilizarse la referencia `this` para pasar una referencia al objeto actual a un método de otro objeto, para su procesamiento.

Tip de rendimiento 8.1

Java conserva el almacenamiento al mantener sólo una copia de cada método por clase; este método es invocado por todos los objetos de esa clase. Por otro lado, cada uno de los objetos tiene su propia copia de las variables de instancia de la clase. Cada método de la clase utiliza a `this` para determinar el objeto específico de la clase a manipular.

Al compilar este programa, el compilador produce dos archivos de clase: uno para la clase `HoraSimple` y otro para la clase `PruebaThis`. Cada clase `public` de Java tiene su propio archivo de clase con una extensión `.class`. El compilador coloca a estos dos archivos de clase en el mismo directorio.

8.6 Inicialización de los objetos de una clase: Constructores

Recuerde que en la sección 8.2 vimos que un *constructor* inicializa las variables de instancia de una clase. Cuando se crea un objeto de una clase, `new` llama al *constructor* de la clase para llevar a cabo la inicialización. Un constructor debe tener el mismo nombre que el de su clase (incluyendo las mismas letras en mayúsculas y minúsculas). Un constructor no puede especificar un tipo o un valor de retorno. Una clase puede contener *constructores sobrecargados*, los cuales permiten a los objetos de esa clase inicializarse de distintas maneras.

Cuando un programa crea la instancia de un objeto de cierta clase, el programa puede proporcionar *inicializadores* (argumentos) entre paréntesis, a la derecha del nombre de la clase. Estos inicializadores se pasan como argumentos al constructor de la clase. Mostraremos esta técnica en la figura 8.6. Anteriormente ya hemos visto esta técnica al crear nuevos objetos de clases como `DecimalFormat`, `JLabel`, `JTextField`, `JTextArea` y `JButton`. Para cada una de estas clases, hemos visto expresiones de creación de instancias de clases de la forma:

```
new NombreDeLaClase( argumento1, argumento2, ..., argumentoN )
```

en donde `new` indica que se creará un nuevo objeto, *NombreDeLaClase* indica el tipo del nuevo objeto, los argumentos especifican los valores utilizados por el constructor de la clase para inicializar el objeto, y toda la expresión completa devuelve una referencia al nuevo objeto.

Se requiere que toda clase tenga cuando menos un constructor. Por lo tanto, si no se declaran constructores para una clase, el compilador crea un *constructor predeterminado* que no toma argumentos. El constructor

predeterminado para una clase llama al constructor sin argumentos para su superclase (la clase que extiende), y luego procede a inicializar las variables de instancia con los valores iniciales especificados en sus declaraciones, o con sus valores predeterminados (cero para los tipos numéricos primitivos, `false` para los valores `boolean` y `null` para las referencias). Si la clase que es extendida por esta clase no tiene un constructor sin argumentos, el compilador genera un mensaje de error. También es posible para el programador proporcionar un constructor sin argumentos. De hecho, el constructor para la clase `Tiempo1` de la figura 8.1 es un constructor sin argumentos. Si un programador declara uno o más constructores para una clase, Java no creará un constructor predeterminado para esa clase.



Buena práctica de programación 8.5

Cada vez que sea apropiado (casi siempre), proporcione un constructor para asegurar que todas las variables de instancia de un objeto se inicialicen correctamente, con valores significativos.



Error común de programación 8.5

Si una clase tiene constructores, pero ninguno de los constructores `public` es constructor sin argumentos, y un programa trata de llamar a un constructor sin argumentos para inicializar un objeto de la clase, se produce un error de compilación. Un constructor puede llamarse sin argumentos solamente si no hay constructores para la clase (se llama al constructor predeterminado) o si hay un constructor `public` sin argumentos.

8.7 Uso de constructores sobrecargados

Como dijimos en la sección 8.6, los constructores de una clase pueden *sobrecargarse*. Los constructores sobrecargados permiten a los objetos de una clase inicializarse de distintas formas. Para sobrecargar constructores, simplemente hay que proporcionar varias declaraciones del constructor con distintas listas de parámetros. Recuerde que en la sección 6.11 vimos que los métodos sobrecargados *deben* tener distintas listas de parámetros.



Error común de programación 8.6

Tratar de sobrecargar un constructor con otro constructor que tenga exactamente la misma firma (nombre y parámetros) es un error de sintaxis.

El constructor de `Tiempo1` en las líneas 12 a 15 de la figura 8.1 inicializa a `hora`, `minuto` y `segundo` con `0` (que es medianoche en formato de hora universal), llamando al método `establecerHora` de la clase. Sin embargo, este constructor no permite a los clientes de la clase inicializar la hora con valores específicos. La clase `Tiempo2` (figura 8.5) contiene cinco constructores sobrecargados que proporcionan formas convenientes de inicializar objetos de la nueva clase `Tiempo2`. Cada constructor garantiza que todo objeto que inicialice empezará en un estado consistente. En este programa, cuatro de los constructores invocan a un quinto constructor, el cual a su vez llama al método `establecerHora` para asegurar que el valor proporcionado para `hora` se encuentre en el rango de 0 a 23 y que los valores para `minuto` y `segundo` se encuentren en el rango de 0 a 59. Si un valor está fuera de rango, se establece en cero mediante el método `establecerHora` (asegurando una vez más que cada variable de instancia permanezca en un estado consistente). Para invocar al constructor apropiado, se comparan el número y tipo de los argumentos especificados en la llamada al constructor con el número y tipos de los parámetros especificados en la declaración de cada constructor. Esto se conoce también como *igualar las firmas*. En la figura 8.6 se utiliza la clase `Tiempo2` para mostrar sus constructores.

```
1  // Fig. 8.5: Tiempo2.java
2  // Declaración de la clase Tiempo2 con constructores sobrecargados.
3  import java.text.DecimalFormat;
4
5  public class Tiempo2 {
6     private int hora;     // 0 - 23
7     private int minuto;   // 0 - 59
8     private int segundo;  // 0 - 59
9
```

Figura 8.5 La clase `Tiempo2` con constructores sobrecargados. (Parte 1 de 3.)

http://librosysolucionarios.net


```
10    // El constructor de Tiempo2 inicializa cada variable de instancia en cero;
11    // asegura que el objeto Tiempo2 empiece en un estado consistente
12    public Tiempo2()
13    {
14       this( 0, 0, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
15    }
16
17    // Constructor de Tiempo2: se proporciona hora; minuto y segundo con
         valor predeterminado de 0
18    public Tiempo2( int h )
19    {
20       this( h, 0, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
21    }
22
23    // Constructor de Tiempo2: se proporcionan hora y minuto; segundo con
         valor predeterminado de 0
24    public Tiempo2( int h, int m )
25    {
26       this( h, m, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
27    }
28
29    // Constructor de Tiempo2: se proporcionan hora, minuto y segundo
30    public Tiempo2( int h, int m, int s )
31    {
32       establecerHora( h, m, s ); // invoca a establecerHora para validar
         la hora
33    }
34
35    // Constructor de Tiempo2: se proporciona otro objeto Tiempo2
36    public Tiempo2( Tiempo2 tiempo )
37    {
38       // invocar al constructor de Tiempo2 con tres argumentos
39       this( tiempo.hora, tiempo.minuto, tiempo.segundo );
40    }
41
42    // establecer un nuevo valor de tiempo, utilizando la hora universal;
         realizar
43    // comprobaciones de validez en los datos; establecer valores inválidos
         en cero
44    public void establecerHora( int h, int m, int s )
45    {
46       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
47       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
48       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
49    }
50
51    // convertir a String en formato de hora universal
52    public String aStringUniversal()
53    {
54       DecimalFormat dosDigitos = new DecimalFormat( "00" );
55
56       return dosDigitos.format( hora ) + ":" +
57          dosDigitos.format( minuto ) + ":" + dosDigitos.format( segundo );
58    }
59
```

Figura 8.5 La clase `Tiempo2` con constructores sobrecargados. (Parte 2 de 3.)

```
60    // convertir a String en formato de hora estándar
61    public String aStringEstandar()
62    {
63       DecimalFormat dosDigitos = new DecimalFormat( "00" );
64
65       return ( (hora == 12 || hora == 0) ? 12 : hora % 12 ) + ":" +
66          dosDigitos.format( minuto ) + ":" + dosDigitos.format( segundo ) +
67          ( hora < 12 ? " AM" : " PM" );
68    }
69
70 } // fin de la clase Tiempo2
```

Figura 8.5 La clase `Tiempo2` con constructores sobrecargados. (Parte 3 de 3.)

La mayor parte del código en la clase `Tiempo2` es idéntico al de la clase `Tiempo1`, por lo que nos concentraremos en los nuevos constructores. En las líneas 12 a 15 se declara el constructor sin argumentos (predeterminado) y se introduce un uso especial de la referencia `this`, la cual se permite sólo como la primera instrucción en el cuerpo de un constructor. En la línea 14 se utiliza a `this` en la sintaxis de llamada al método, para invocar al constructor de `Tiempo2` que toma tres argumentos (declarados en las líneas 30 a 33). El constructor sin argumentos pasa valores de 0 para la hora, el minuto y el segundo. Al utilizar la referencia `this` como se muestra aquí podemos reutilizar el código de inicialización proporcionado por otro constructor, en vez de tener que definir código similar o idéntico en el cuerpo del constructor sin argumentos.



Error común de programación 8.7

Es un error de sintaxis cuando la referencia `this` se utiliza en el cuerpo de un constructor para llamar a otro constructor de la misma clase, y esa instrucción no es la primera en el constructor. También es un error de sintaxis cuando un método no constructor intenta invocar a un constructor directamente mediante la referencia `this`.

En las líneas 18 a 21 se declara un constructor de `Tiempo2` que recibe un solo argumento `int`, el cual representa a la `hora`, que se pasa junto con los valores de 0 para `minuto` y `segundo` al constructor en las líneas 30 a 33. En las líneas 24 a 27 se declara un constructor `Tiempo2` que recibe dos argumentos `int`, los cuales representan la `hora` y el `minuto`, que se pasan junto con el valor de 0 para `segundo` al constructor en las líneas 30 a 33. En las líneas 30 a 33 se declara el constructor `Tiempo2` que recibe tres argumentos `int`, los cuales representan a la `hora`, el `minuto` y el `segundo`. Este constructor llama a `establecerHora` para llevar a cabo la inicialización de las variables de instancia.



Error común de programación 8.8

Un constructor puede llamar a otros métodos de la clase. Esté consciente que las variables de estado tal vez no se encuentren aún en un estado consistente, ya que el constructor está en el proceso de inicializar el objeto. Utilizar variables de instancia antes de que hayan sido inicializadas apropiadamente es un error lógico.

En las líneas 36 a 40 se declara un constructor `Tiempo2` que recibe una referencia `Tiempo2` a otro objeto `Tiempo2`. En este caso, los valores del argumento `Tiempo2` se pasan al constructor en las líneas 30 a 33 para inicializar la `hora`, el `minuto` y el `segundo`. Observe que este constructor accede directamente a los valores de `hora`, `minuto` y `segundo` de su argumento `tiempo`. A pesar de que sabemos que `hora`, `minuto` y `segundo` se declaran como variables `private` de la clase `Tiempo2`, podemos acceder a estos valores con las expresiones `tiempo.hora`, `tiempo.minuto` y `tiempo.segundo` (línea 39). Esto se debe a una relación especial entre los objetos de la misma clase.



Observación de ingeniería de software 8.6

Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos del segundo objeto (incluyendo a los que son `private`).

Ninguno de los constructores especifica un tipo de valor de retorno. (Recuerde, esto no está permitido para los constructores.) Además, todos los constructores reciben distintos números de argumentos y/o distintos tipos

de argumentos. A pesar de que sólo dos de los constructores reciben valores para la `hora`, `minuto` y `segundo`, todos los constructores especifican valores para `hora`, `minuto` y `segundo` utilizando ceros para sustituir los valores faltantes.

La clase `PruebaTiempo3` (figura 8.6) crea seis objetos `Tiempo2` (líneas 9 a 14) para mostrar cómo invocar a los distintos constructores de `Tiempo2`. En la línea 9 se muestra que el constructor sin argumentos se invoca colocando un conjunto vacío de paréntesis después del nombre de la clase, al asignar un objeto `Tiempo2` mediante `new`. En las líneas 10 a 14 del programa se muestra el paso de argumentos a los constructores de `Tiempo2`. El constructor apropiado se invoca comparando el número y tipos de los argumentos especificados en la llamada al constructor, con el número y tipos de los parámetros especificados en la declaración de cada constructor. Entonces, en la línea 10 se invoca al constructor de las líneas 18 a 21 de la figura 8.5. En la línea 11 se invoca al constructor de las líneas 24 a 27 de la figura 8.5. En las líneas 12 y 13 se invoca el constructor de las líneas 30 a 33 de la figura 8.5. En la línea 14 se invoca al constructor de las líneas 36 a 40 de la figura 8.5. El resto de la aplicación crea una cadena que contiene las representaciones en cadena de cada objeto `Tiempo2` inicializado.

```
1  // Fig. 8.6: PruebaTiempo3.java
2  // Constructores sobrecargados utilizados para inicializar objetos Tiempo2.
3  import javax.swing.*;
4
5  public class PruebaTiempo3 {
6
7     public static void main( String args[] )
8     {
9        Tiempo2 t1 = new Tiempo2();                  // 00:00:00
10       Tiempo2 t2 = new Tiempo2( 2 );               // 02:00:00
11       Tiempo2 t3 = new Tiempo2( 21, 34 );          // 21:34:00
12       Tiempo2 t4 = new Tiempo2( 12, 25, 42 );      // 12:25:42
13       Tiempo2 t5 = new Tiempo2( 27, 74, 99 );      // 00:00:00
14       Tiempo2 t6 = new Tiempo2( t4 );              // 12:25:42
15
16       String salida = "Se construyó con: " +
17          "\nt1: todos los argumentos predeterminados" +
18          "\n       " + t1.aStringUniversal() +
19          "\n       " + t1.aStringEstandar();
20
21       salida += "\nt2: se especificó hora; minuto y segundo predeterminados" +
22          "\n       " + t2.aStringUniversal() +
23          "\n       " + t2.aStringEstandar();
24
25       salida += "\nt3: se especificaron hora y minuto; segundo predeterminado" +
26          "\n       " + t3.aStringUniversal() +
27          "\n       " + t3.aStringEstandar();
28
29       salida += "\nt4: se especificaron hora, minuto y segundo" +
30          "\n       " + t4.aStringUniversal() +
31          "\n       " + t4.aStringEstandar();
32
33       salida += "\nt5: se especificaron todos como valores inválidos" +
34          "\n       " + t5.aStringUniversal() +
35          "\n       " + t5.aStringEstandar();
36
```

Figura 8.6 Constructores sobrecargados utilizados para inicializar objetos `Tiempo2`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
37          salida += "\nt6: se especificó objeto t4 de la clase Tiempo2" +
38             "\n       " + t6.aStringUniversal() +
39             "\n       " + t6.aStringEstandar();
40
41          JOptionPane.showMessageDialog( null, salida,
42             "Constructores sobrecargados", JOptionPane.INFORMATION_MESSAGE );
43
44          System.exit( 0 );
45
46       } // fin de main
47
48    } // fin de la clase PruebaTiempo3
```

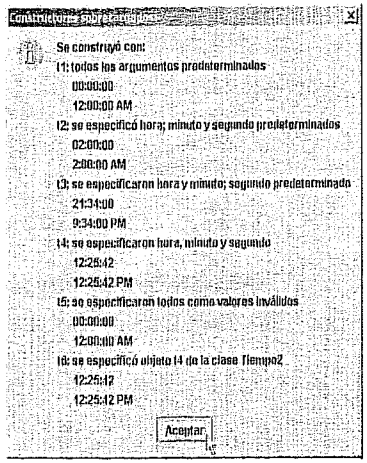


Figura 8.6 Constructores sobrecargados utilizados para inicializar objetos `Tiempo2`. (Parte 2 de 2.)

Cada constructor de `Tiempo2` en la figura 8.5 podría escribirse de manera que incluyera una copia de las instrucciones apropiadas del método `establecerHora`. Esto podría ser ligeramente más eficiente, ya que se eliminarían la llamada adicional al constructor y la llamada a `establecerHora`. Sin embargo, considere cambiar la representación de la hora, de tres valores `int` (que requieren 12 bytes de memoria) a un solo valor `int` para representar el número total de segundos transcurridos en el día (que requiere 4 bytes de memoria). Al codificar los constructores de `Tiempo2` y el método `establecerHora` en forma idéntica se dificulta dicho cambio en esta declaración de clase. Si la implementación del método `establecerHora` cambiara, la implementación de los constructores de `Tiempo2` tendría que cambiar de igual forma. Para hacer que los constructores de `Tiempo2` llamen al constructor con tres argumentos (o incluso que llamen a `establecerHora` directamente) se requiere que cualquier cambio a la implementación de `establecerHora` se realice sólo una vez. Esto reduce la posibilidad de un error de programación al alterar la implementación.



Observación de ingeniería de software 8.7

Si el método de una clase proporciona la funcionalidad requerida por un constructor (u otro método) de la clase, llame a ese método desde el constructor (u otro método). Esto simplifica el mantenimiento del código y reduce la posibilidad de introducir errores en el código.

8.8 Uso de los métodos *set (establecer)* y *get (obtener)*

Los campos privados pueden manipularse solamente mediante métodos de la clase en donde están declarados. Una manipulación típica sería el ajuste del saldo bancario de un cliente (por ejemplo, una variable de instancia

http://librosysolucionarios.net

`private` de una clase llamada `CuentaBancaria`) mediante un método llamado `calcularInteres`. Las clases a menudo proporcionan métodos `public` para permitir a los clientes de la clase *establecer* (es decir, asignar valores a) u *obtener* (es decir, recibir los valores de) variables de instancia `private`. No es necesario llamar a estos métodos *establecer* y *obtener*, pero a menudo así sucede.

Como ejemplo de nomenclatura, un método para establecer la variable de instancia `tasaInteres` se llamaría típicamente `establecerTasaInteres`, y un método para obtener la `tasaDeInteres` se llamaría típicamente `obtenerTasaInteres`. Los métodos *establecer* también se conocen comúnmente como *métodos mutadores* (porque generalmente cambian un valor). Los métodos *obtener* también se conocen comúnmente como *métodos de acceso* o *métodos de consulta*.

Parece ser que proporcionar herramientas para *establecer* y *obtener* es esencialmente lo mismo que hacer las variables de instancia públicas. Ésta es una sutileza de Java que hace del lenguaje algo tan deseable para la ingeniería de software. Si una variable de instancia se declara como `public`, cualquier método que tenga una referencia a un objeto de la clase en la que se declare esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como `private`, un método *obtener* `public` evidentemente parece permitir a otros métodos el acceso a la variable; sin embargo, el método *obtener* controla la manera en que el cliente puede tener acceso a la variable. Por ejemplo, si representamos la hora como el número total de segundos en un día, un método `obtenerHora` calcularía la hora actual y devolvería ese valor. El código cliente no necesita saber que no hay una variable de instancia llamada `hora` en el objeto. El cliente simplemente se preocupa por el(los) servicio(s) que proporciona el objeto. Un método *establecer* público puede (y debe) escudriñar cuidadosamente los intentos por modificar el valor de la variable. Esto asegura que el nuevo valor sea apropiado para ese elemento de datos. Por ejemplo, un intento por *establecer* el día de la semana en una fecha 37 sería rechazado, un intento por *establecer* el peso de una persona en un valor negativo sería rechazado, y así sucesivamente. Entonces, aunque los métodos *establecer* y *obtener* pueden proporcionar acceso a los datos privados, el programador restringe su acceso mediante la implementación de los métodos.

Los beneficios de la integridad de los datos no se dan automáticamente sólo porque las variables de instancia se declaren como `private`; el programador debe proporcionar la comprobación de su validez. Java permite a los programadores diseñar mejores programas de una manera conveniente. Los métodos *establecer* de una clase pueden devolver valores que indiquen que hubo intentos de asignar datos inválidos a los objetos de la clase. Esto permite a los clientes de la clase probar los valores de retorno de los métodos *establecer*, para determinar si los objetos que están manipulando son válidos, y para tomar la acción apropiada si los objetos no son válidos. En el capítulo 15, mostraremos una manera más robusta en la que los clientes de una clase pueden ser notificados si un objeto no es válido.



Observación de ingeniería de software 8.8

Los métodos que establecen los valores de variables `private` deben verificar que los nuevos valores deseados sean apropiados; si no lo son, los métodos establecer deben colocar a las variables `private` en un estado consistente apropiado.



Observación de ingeniería de software 8.9

Todas las variables de instancia de una clase deben mantenerse como private. Cuando sea necesario, proporcione métodos public para cambiar los valores de las variables de instancia private y para recuperar sus valores. Esta arquitectura ayuda a ocultar la implementación de una clase de sus clientes, lo cual mejora la capacidad de modificación de los programas.



Observación de ingeniería de software 8.10

Los diseñadores de clases no necesitan proporcionar métodos set o get para cada campo privado; estas capacidades deben proporcionarse solamente cuando esto tenga sentido.



Tip para prevenir errores 8.2

Al hacer que las variables de instancia de una clase sean `private` y que los métodos de la clase sean `public` se facilita la depuración, ya que los problemas con la manipulación de los datos se localizan en los métodos de la clase.

El applet de las figuras 8.7 (clase `Tiempo3`) y 8.8 (clase `PruebaTiempo4`) mejora nuestra clase `Hora` (ahora llamada `Tiempo3`), incluyendo métodos *establecer* y *obtener* para las variables de instancia privadas `hora`, `minuto` y `segundo`. Los métodos *establecer* controlan estrictamente la asignación de valores válidos

a las variables de instancia. Los intentos por establecer cualquier variable de instancia en un valor incorrecto hacen que esta variable se establezca en cero (dejando así a la variable de instancia en un estado consistente). Cada método *obtener* devuelve el valor de la variable de instancia apropiada. En este applet presentamos también técnicas mejoradas para el manejo de eventos de GUI, para que usted pueda probar los nuevos métodos *establecer* y *obtener* de la clase **Tiempo3** en forma interactiva.

```
1  // Fig. 8.7: Tiempo3.java
2  // Declaración de la clase Tiempo3 con métodos establecer y obtener.
3  import java.text.DecimalFormat;
4
5  public class Tiempo3 {
6     private int hora;       // 0 - 23
7     private int minuto;     // 0 - 59
8     private int segundo;    // 0 - 59
9
10    // El constructor de Tiempo3 inicializa cada variable de instancia en cero;
11    // asegura que el objeto Tiempo3 inicie en un estado consistente
12    public Tiempo3()
13    {
14       this( 0, 0, 0 ); // invoca al constructor de Tiempo3 con tres argumentos
15    }
16
17    // Constructor de Tiempo3: se proporciona hora; minuto y segundo con
         valor predeterminado de 0
18    public Tiempo3( int h )
19    {
20       this( h, 0, 0 ); // invoca al constructor de Tiempo3 con tres argumentos
21    }
22
23    // Constructor de Tiempo3: se proporcionan hora y minuto; segundo con
         valor predeterminado de 0
24    public Tiempo3( int h, int m )
25    {
26       this( h, m, 0 ); // invoca al constructor de Tiempo3 con tres argumentos
27    }
28
29    // Constructor de Tiempo3: se proporcionan hora, minuto y segundo
30    public Tiempo3( int h, int m, int s )
31    {
32       establecerHora( h, m, s );
33    }
34
35    // Constructor de Tiempo3: se proporciona otro objeto Tiempo3
36    public Tiempo3( Tiempo3 tiempo )
37    {
38       // invocar al constructor de Tiempo3 con tres argumentos
39       this( tiempo.obtenerHora(), tiempo.obtenerMinuto(),
         tiempo.obtenerSegundo() );
40    }
41
42    // Métodos establecer
43    // establecer un nuevo valor de hora usando hora universal; realizar
44    // comprobaciones de validez en los datos; establecer valores inválidos
         en cero
```

Figura 8.7 La clase **Tiempo3** con métodos *establecer* y *obtener*. (Parte 1 de 3.)

http://librosysolucionarios.net


```
45    public void establecerHora( int h, int m, int s )
46    {
47       establecerHora( h );    // establecer la hora
48       establecerMinuto( m );  // establecer el minuto
49       establecerSegundo( s ); // establecer el segundo
50    }
51
52    // validar y establecer la hora
53    public void establecerHora( int h )
54    {
55       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
56    }
57
58    // validar y establecer minuto
59    public void establecerMinuto( int m )
60    {
61       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
62    }
63
64    // validar y establecer segundo
65    public void establecerSegundo( int s )
66    {
67       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
68    }
69
70    // Métodos obtener
71    // obtener el valor de hora
72    public int obtenerHora()
73    {
74       return hora;
75    }
76
77    // obtener el valor de minuto
78    public int obtenerMinuto()
79    {
80       return minuto;
81    }
82
83    // obtener el valor de segundo
84    public int obtenerSegundo()
85    {
86       return segundo;
87    }
88
89    // convertir a String en formato de hora universal
90    public String aStringUniversal()
91    {
92       DecimalFormat dosDigitos = new DecimalFormat( "00" );
93
94       return dosDigitos.format( obtenerHora() ) + ":" +
95          dosDigitos.format( obtenerMinuto() ) + ":" +
96          dosDigitos.format( obtenerSegundo() );
97    }
98
99    // convertir a String en formato de hora estándar
```

Figura 8.7 La clase **Tiempo3** con métodos *establecer* y *obtener*. (Parte 2 de 3.)

```
100    public String aStringEstandar()
101    {
102       DecimalFormat dosDigitos = new DecimalFormat( "00" );
103
104       return ( ( obtenerHora() == 12 || obtenerHora() == 0 ) ?
105          12 : obtenerHora() % 12 ) + ":" + dosDigitos.format(
          obtenerMinuto() ) +
106          ":" + dosDigitos.format( obtenerSegundo() ) +
107          ( obtenerHora() < 12 ? " AM" : " PM" );
108    }
109
110 } // fin de la clase Tiempo3
```

Figura 8.7 La clase **Tiempo3** con métodos *establecer* y *obtener*. (Parte 3 de 3.)

Los nuevos métodos *establecer* de la clase **Tiempo3** se declaran en la figura 8.7, en las líneas 53 a 56, 59 a 62 y 65 a 68. Cada método ejecuta la misma instrucción condicional que se encontraba anteriormente en el método **establecerHora** para ajustar las variables **hora**, **minuto** o **segundo**. Estos nuevos métodos nos permiten redefinir el cuerpo del método **establecerHora** para mejorar la ingeniería de software. Recuerde que, si el método de una clase ya proporciona toda o parte de la funcionalidad requerida por otro método de la clase, llame a ese método. El método **establecerHora** (líneas 45 a 50) ahora llama a los métodos **establecerHora**, **establecerMinuto** y **establecerSegundo**, cada uno de los cuales realiza parte de la tarea de **establecerHora**.

Los nuevos métodos *get* de la clase se declaran en las líneas 72 a 75, 78 a 81 y 84 a 87. Cada método devuelve el valor de **hora**, **minuto** o **segundo**. Se devuelve una copia de cada valor, ya que éstas son todas variables de tipo primitivo. Utilizando estos nuevos métodos, modificamos los cuerpos de los métodos **aStringUniversal** (líneas 90 a 97) y **aStringEstandar** (líneas 100 a 108). En ambos casos, cada uso de las variables de instancia **hora**, **minuto** o **segundo** se reemplaza con una llamada a **establecerHora**, **establecerMinuto** y **establecerSegundo**.

Los métodos *establecer* y *obtener* tienen una importante ventaja de ingeniería de software: la flexibilidad. Al utilizar métodos *establecer* y *obtener* a lo largo de los constructores y demás métodos de la clase **Tiempo3**, minimizamos los cambios que debemos realizar en la declaración de la clase en caso de que alteremos la representación de los datos. Por ejemplo, podríamos cambiar la representación de los datos de **hora**, **minuto** y **segundo** por otra representación, como un solo entero para representar el total de segundos transcurridos en el día. Al realizar dichos cambios, debemos modificar cada una de las partes de la clase que tengan acceso directo a **hora**, **minuto** o **segundo**. Si utilizamos los métodos *establecer* y *obtener* de la clase para acceder a **hora**, **minuto** y **segundo** a lo largo de la clase, entonces sólo necesitamos proporcionar una nueva lógica para los métodos *establecer* y *obtener*. El utilizar esta técnica también permite a los programadores cambiar la implementación de una clase sin afectar a los clientes de esa clase (siempre y cuando las llamadas a todos los métodos **public** se sigan realizando de la misma manera).



Observación de ingeniería de software 8.11

El acceso a los datos **private** *a través de métodos de acceso* establecer y obtener *no sólo protege a las variables de instancia para que no reciban valores inválidos, sino que también oculta la representación interna de las variables de instancia a los clientes de esa clase. Por lo tanto, si la representación de los datos cambia (generalmente para reducir la cantidad de almacenamiento requerido, o para mejorar el rendimiento), sólo tienen que cambiar las implementaciones de los métodos; las implementaciones del cliente no necesitan cambiar, siempre y cuando se preserve la interfaz proporcionada por los métodos.*

El subprograma **PruebaTiempo4** (figura 8.8) proporciona una interfaz gráfica de usuario que permite al usuario probar los métodos de la clase **Tiempo3**. El usuario puede establecer el valor de la hora, el minuto o el segundo al escribir un valor en el objeto **JTextField** apropiado, y después oprimir *Intro*. El usuario puede también hacer clic en el botón **Agregar 1 a Segundo** para incrementar la hora por un segundo. El libro de eventos mostrado en este ejemplo es similar al que se muestra en las figuras 6.9 y 6.16. Todos los eventos

http://librosysolucionarios.net

para los objetos **JTextField** y **JButton** en este applet se procesan en el método **actionPerformed** (líneas 64 a 90). Observe que en las líneas 54 a 57 se llama a **addActionListener** para indicar que el applet debe empezar a escuchar eventos de los objetos **JTextField** llamados **horaCampo**, **minutoCampo** y **segundoCampo**, y del objeto **JButton** llamado **ticBoton**. Las cuatro llamadas utilizan a **this** como argumento, indicando que el método **actionPerformed** de nuestro objeto del applet **PruebaTiempo4** debe invocarse para cada interacción del usuario con estos cuatro componentes de GUI. En ejemplos anteriores con GUI, sólo un componente de la GUI generaba eventos. Esto produce una interesante pregunta: ¿cómo determinamos cuál fue el componente de la GUI con el que interactuó el usuario?

En **actionPerformed**, **evento.getSource()** (líneas 67, 71, 77 y 83) determina cuál fue el componente de la GUI que generó el evento. Por ejemplo, en la línea 67 se determina si **ticBoton** fue oprimido por el usuario. De ser así, se ejecuta el cuerpo de la instrucción **if**. En caso contrario, el programa evalúa la condición en la instrucción **if** de la línea 71, y así sucesivamente. Todo evento tiene un *origen*: el componente de la GUI con el que interactuó el usuario para indicar al programa que realice una tarea. El parámetro **ActionEvent** contiene una referencia al origen del evento. La condición en la línea 67 pregunta, "¿es **ticBoton** el *origen del evento*?" Esta condición compara las referencias en ambos lados del operador **==** para determinar si se refieren al mismo objeto. En este caso, si ambos se refieren al objeto **JButton** llamado **ticBoton**, entonces el programa sabe que el usuario oprimió ese botón. Recuerde que en la sección 6.8 vimos que el origen del evento llama a **actionPerformed** en respuesta a la interacción del usuario.

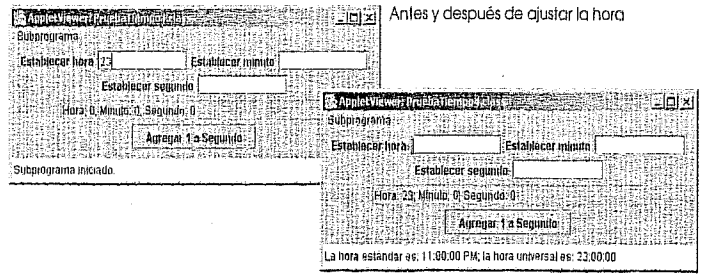
```
1  // Fig. 8.8: PruebaTiempo4.java
2  // Demostración de los métodos establecer y obtener de la clase Tiempo3.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaTiempo4 extends JApplet implements ActionListener {
8     private Tiempo3 tiempo;
9
10    private JLabel horaEtiqueta, minutoEtiqueta, segundoEtiqueta;
11    private JTextField horaCampo, minutoCampo, segundoCampo, pantallaCampo;
12    private JButton ticBoton;
13
14    // crear objeto Tiempo3 y configurar GUI
15    public void init()
16    {
17       tiempo = new Tiempo3(); // crear objeto Tiempo3
18
19       // obtener el panel de contenido del applet y cambiar esquema a
            FlowLayout
20       Container contenedor = getContentPane();
21       contenedor.setLayout( new FlowLayout() );
22
23       // configurar horaEtiqueta y horaCampo
24       horaEtiqueta = new JLabel( "Establecer hora" );
25       horaCampo = new JTextField( 10 );
26       contenedor.add( horaEtiqueta );
27       contenedor.add( horaCampo );
28
29       // configurar minutoEtiqueta y minutoCampo
30       minutoEtiqueta = new JLabel( "Establecer minuto" );
31       minutoCampo = new JTextField( 10 );
32       contenedor.add( minutoEtiqueta );
33       contenedor.add( minutoCampo );
```

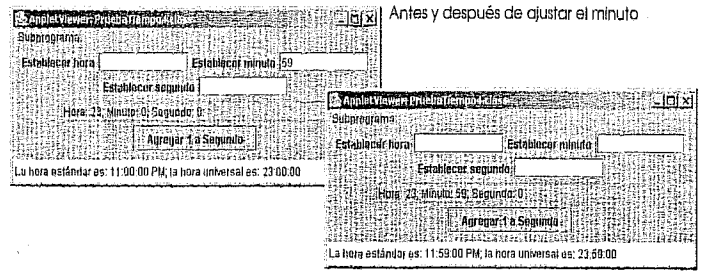
Figura 8.8 Métodos *establecer* y *obtener* utilizados para manipular un objeto **Tiempo3**. (Parte 1 de 4.)

```
34
35       // configurar segundoEtiqueta y segundoCampo
36       segundoEtiqueta = new JLabel( "Establecer segundo" );
37       segundoCampo = new JTextField( 10 );
38       contenedor.add( segundoEtiqueta );
39       contenedor.add( segundoCampo );
40
41       // configurar pantallaCampo
42       pantallaCampo = new JTextField( 30 );
43       pantallaCampo.setEditable( false );
44       contenedor.add( pantallaCampo );
45
46       // configurar ticBoton
47       ticBoton = new JButton( "Agregar 1 a Segundo" );
48       contenedor.add( ticBoton );
49
50       // registrar los manejadores de eventos; este applet es el
            ActionListener,
51       // el cual contiene el método actionPerformed que será llamado para
52       // manejar los eventos de acción generados por horaCampo, minutoCampo,
53       // segundoCampo y ticBoton
54       horaCampo.addActionListener( this );
55       minutoCampo.addActionListener( this );
56       segundoCampo.addActionListener( this );
57       ticBoton.addActionListener( this );
58
59       mostrarHora(); // actualizar texto en pantallaCampo y barra de estado
60
61    } // fin del método init
62
63    // manejador de eventos para los eventos de botón y campo de texto
64    public void actionPerformed( ActionEvent evento )
65    {
66       // procesar evento de ticBoton
67       if ( evento.getSource() == ticBoton )
68          tic();
69
70       // procesar evento de horaCampo
71       else if ( evento.getSource() == horaCampo ) {
72          tiempo.establecerHora( Integer.parseInt( evento.getActionCommand() ) );
73          horaCampo.setText( "" );
74       }
75
76       // procesar evento de minutoCampo
77       else if ( evento.getSource() == minutoCampo ) {
78          tiempo.establecerMinuto( Integer.parseInt( evento.getActionCommand() ) );
79          minutoCampo.setText( "" );
80       }
81
82       // procesar evento de segundoCampo
83       else if ( evento.getSource() == segundoCampo ) {
84          tiempo.establecerSegundo( Integer.parseInt( evento.getActionCommand() ) );
85          segundoCampo.setText( "" );
86       }
87
```

Figura 8.8 Métodos *establecer* y *obtener* utilizados para manipular un objeto `Tiempo3`. (Parte 2 de 4.)


```
88       mostrarHora(); // actualizar texto en pantallaCampo y barra de estado
89
90    } // fin del método actionPerformed
91
92    // actualizar pantallaCampo y barra de estado del contenedor de applets
93    public void mostrarHora()
94    {
95       pantallaCampo.setText( "Hora: " + tiempo.obtenerHora() + "; Minuto: " +
96          tiempo.obtenerMinuto() + "; Segundo: " + tiempo.obtenerSegundo() );
97
98       showStatus( "La hora estándar es: " + tiempo.aStringEstandar() +
99          "; la hora universal es: " + tiempo.aStringUniversal() );
100
101   } // fin del método mostrarHora
102
103   // sumar uno al segundo y actualizar la hora/minuto si es necesario
104   public void tic()
105   {
106      tiempo.establecerSegundo( ( tiempo.obtenerSegundo() + 1 ) % 60 );
107
108      if ( tiempo.obtenerSegundo() == 0 ) {
109         tiempo.establecerMinuto( ( tiempo.obtenerMinuto() + 1 ) % 60 );
110
111         if ( tiempo.obtenerMinuto() == 0 )
112            tiempo.establecerHora( ( tiempo.obtenerHora() + 1 ) % 24 );
113      }
114
115   } // fin del método tic
116
117 } // fin de la clase PruebaTiempo4
```





http://librosysolucionarios.net

Figura 8.8 Métodos *establecer* y *obtener* utilizados para manipular un objeto `Tiempo3`. (Parte 3 de 4.)

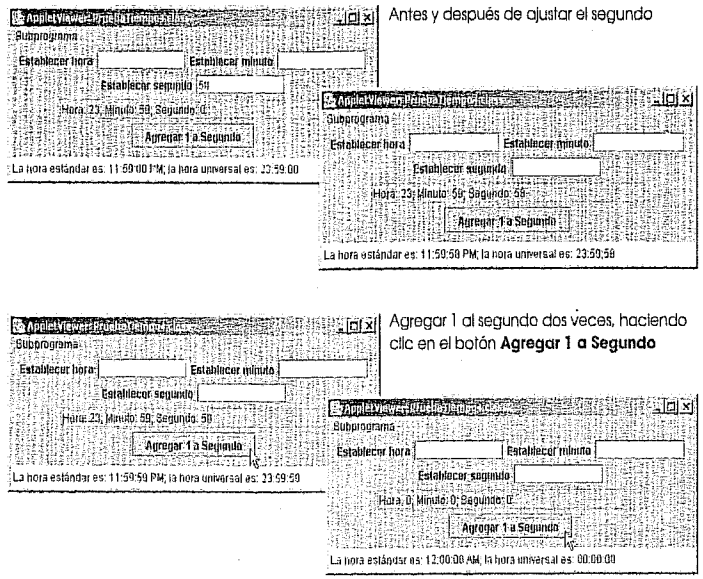


Figura 8.8 Métodos *establecer* y *obtener* utilizados para manipular un objeto `Tiempo3`. (Parte 4 de 4.)

Después de procesar cada uno de los eventos de la interfaz de usuario, en la línea 88 se hace una llamada a `mostrarHora` para mostrar la nueva hora como una cadena en la barra de estado del applet, y los valores de la hora, minuto y segundo en el campo de texto `pantallaCampo`. En las ventanas de salida de la figura 8.8 se muestra el applet antes y después de las siguientes operaciones: ajuste de la hora a 23, ajuste del minuto a 59, ajuste del segundo a 58 e incremento del valor de los segundos dos veces mediante el botón `Agregar 1 a Segundo`.

Cuando el usuario hace clic en el botón `Agregar 1 a Segundo` (como se muestra en las últimas dos ventanas de salida de la figura 8.8), el método `actionPerformed` llama al método `tic` (declarado en las líneas 104 a 115) del applet. El método `tic` utiliza los nuevos métodos *establecer* y *obtener* para incrementar el segundo apropiadamente.

8.9 Composición

Una clase puede tener como miembros referencias a objetos de otras clases. A esta característica se le conoce como *composición*. Por ejemplo, un objeto de la clase `RelojAlarma` necesita saber cuándo se supone que debe sonar su alarma, así que ¿por qué no incluir una referencia a un objeto `Tiempo` como miembro del objeto `RelojAlarma`?



Observación de ingeniería de software 8.12

Una forma de reutilización de software es la composición, en donde una clase tiene como miembros referencias a objetos de otras clases.

El siguiente programa contiene tres clases: `Fecha` (figura 8.9), `Empleado` (figura 8.10) y `PruebaEmpleado` (figura 8.11). La clase `Empleado` tiene las variables de instancia `primerNombre`, `apellidoPaterno`, `fechaNacimiento` y `fechaContratacion`. Los miembros `fechaNacimiento` y `fechaContratacion` (líneas 7 y 8 de la figura 8.10) son referencias a objetos `Fecha` que tienen las varia-

bles de instancia `mes`, `dia` y `anio`. Esto demuestra que una clase puede tener referencias a objetos de otras clases. Observe que la clase `Empleado` no importa a la clase `Fecha`, ya que ambas se encuentran en el mismo paquete predeterminado.

La clase `PruebaEmpleado` crea una instancia de la clase `Empleado`, inicializa y muestra los valores de sus variables de instancia. El constructor de `Empleado` (figura 8.10, líneas 11 a 18) toma cuatro argumentos: `nombre`, `apellido`, `fechaDeNacimiento` y `fechaDeContratacion`. Los objetos referenciados por estos parámetros se asignan a las variables de instancia correspondientes del objeto `Empleado`.

```
1  // Fig. 8.9: Fecha.java
2  // Declaración de la clase Fecha.
3
4  public class Fecha {
5     private int mes;     // 1-12
6     private int dia;     // 1-31 con base en el mes
7     private int anio;    // cualquier año
8
9     // constructor: llama a comprobarMes para confirmar el valor apropiado de mes;
10    // llama a comprobarDia para confirmar el valor apropiado de dia
11    public Fecha( int elMes, int elDia, int elAnio )
12    {
13       mes = comprobarMes( elMes );    // validar mes
14       anio = elAnio;                  // podría validar anio
15       dia = comprobarDia( elDia );    // validar dia
16
17       System.out.println( "Constructor de objeto Fecha para la fecha " +
18          aStringFecha() );
19
20    } // fin del constructor de Fecha
21
22    // método utilitario para confirmar el valor apropiado del mes
23    private int comprobarMes( int mesPrueba )
24    {
25       if ( mesPrueba > 0 && mesPrueba <= 12 )  // validar mes
26          return mesPrueba;
27
28       else { // mes es inválido
29          System.out.println( "Mes inválido (" + mesPrueba +
30            ") se establece en 1." );
31          return 1;  // mantener el objeto en estado consistente
32       }
33
34    } // fin del método comprobarMes
35
36    // método utilitario para confirmar valor apropiado de dia, con base en mes y anio
37    private int comprobarDia( int diaPrueba )
38    {
39       int diasPorMes[] =
40          { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
41
42       // comprobar si el dia está dentro del rango del mes
43       if ( diaPrueba > 0 && diaPrueba <= diasPorMes[ mes ] )
44          return diaPrueba;
45
```

Figura 8.9 La clase `Fecha`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
46        // comprobar si es año bisiesto
47        if ( mes == 2 && diaPrueba == 29 && ( anio % 400 == 0 ||
48             ( anio % 4 == 0 && anio % 100 != 0 ) ) )
49           return diaPrueba;
50
51        System.out.println( "Día inválido (" + diaPrueba + ") se establece en 1." );
52
53        return 1;  // mantener el objeto en estado consistente
54
55     } // fin del método comprobarDia
56
57     // devolver un String de la forma mes/día/año
58     public String aStringFecha()
59     {
60        return mes + "/" + dia + "/" + anio;
61     }
62
63  } // fin de la clase Fecha
```

Figura 8.9 La clase **Fecha**. (Parte 2 de 2.)

```
1  // Fig. 8.10: Empleado.java
2  // Declaración de la clase Empleado.
3
4  public class Empleado {
5     private String primerNombre;
6     private String apellidoPaterno;
7     private Fecha fechaNacimiento;
8     private Fecha fechaContratacion;
9
10    // constructor para inicializar nombre, fecha de nacimiento y de contratación
11    public Empleado( String nombre, String apellido, Fecha fechaDeNacimiento,
12       Fecha fechaDeContratacion )
13    {
14       primerNombre = nombre;
15       apellidoPaterno = apellido;
16       fechaNacimiento = fechaDeNacimiento;
17       fechaContratacion = fechaDeContratacion;
18    }
19
20    // convertir Empleado a formato de String
21    public String aStringEmpleado()
22    {
23       return apellidoPaterno + ", " + primerNombre +
24          "  Contratado: " + fechaContratacion.aStringFecha() +
25          "  Fecha de nacimiento: " + fechaNacimiento.aStringFecha();
26    }
27
28 } // fin de la clase Empleado
```

Figura 8.10 La clase **Empleado** con referencias a un objeto miembro.

http://librosysolucionarios.net


```
1  // Fig. 8.11: PruebaEmpleado.java
2  // Demostración de un objeto que tiene como miembro a otro objeto.
3  import javax.swing.JOptionPane;
4
5  public class PruebaEmpleado {
6
7     public static void main( String args[] )
8     {
9        Fecha nacimiento = new Fecha( 7, 24, 1949 );
10       Fecha contratacion = new Fecha( 3, 12, 1988 );
11       Empleado empleado = new Empleado( "Juan", "Pérez", nacimiento,
         contratacion );
12
13       JOptionPane.showMessageDialog( null, empleado.aStringEmpleado(),
14          "Prueba de la clase Empleado", JOptionPane.INFORMATION_MESSAGE );
15
16       System.exit( 0 );
17    }
18
19 } // fin de la clase PruebaEmpleado
```




```
Constructor de objeto Fecha para la fecha 7/24/1949
Constructor de objeto Fecha para la fecha 3/12/1990
```

Figura 8.11 Demostración del objeto miembro.

8.10 Recolección de basura

Hemos visto que los constructores son capaces de inicializar las variables de instancia de un objeto cuando éste es creado. Los constructores adquieren varios recursos del sistema, como la memoria. Necesitamos una manera disciplinada de regresar los recursos al sistema cuando ya no son necesarios, para evitar una "fuga de recursos". Java realiza automáticamente una *recolección de basura* para ayudar a regresar al sistema la memoria ocupada por los objetos que ya no se utilizan. Cuando ya no existen más referencias a un objeto, éste queda *marcado para la recolección de basura*. La memoria de ese objeto podrá reclamarse al momento en que se ejecute el *recolector de basura*. Por lo tanto, es menos probable que ocurran en Java las fugas de memoria, que son comunes en lenguajes como C y C++ (debido a que la memoria no se reclama automáticamente en esos lenguajes). Sin embargo, pueden ocurrir otras fugas de recursos.

Toda clase en Java cuenta con un *método finalizador* que regresa los recursos al sistema. El método finalizador se llama para realizar las *tareas de preparación para la terminación* en un objeto, antes de que el recolector de basura reclame la memoria de ese objeto. El finalizador de una clase siempre lleva el nombre **finalize**, no lleva parámetros y tiene el tipo de valor de retorno **void**. Este método no puede sobrecargarse. El método **finalize** se declara en la clase **Object** para guardar un espacio, sin hacer nada. Recuerde que en la sección 8.2 vimos que todas las clases en Java heredan los métodos de la clase **Object**. Esto garantiza que todas las clases tengan un método **finalize** para que el recolector de basura lo pueda llamar.

Observación de ingeniería de software 8.13

No se garantiza que el recolector de basura se ejecute; por lo tanto, no se garantiza que el método **finalize** *de un objeto vaya a llamarse. No diseñe clases que dependan de que el recolector de basura llame al método* **finalize** *de un objeto para regresar recursos al sistema.*

Los finalizadores se utilizan muy raras veces en aplicaciones reales de Java. Principalmente, `finalize` se utiliza en clases que invocan *código nativo* (programas o bibliotecas que no están escritas en Java y que son específicas para el sistema operativo de la computadora en la que se ejecuta el programa de Java). El código nativo está fuera del alcance de este libro. Generalmente no se recomienda declarar un método `finalize` para una clase. Sin embargo, en la sección 8.11 utilizaremos un método `finalize` con el fin de mostrar cuándo se ejecuta el recolector de basura.

8.11 Miembros de clase estáticos

Cada objeto tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe compartirse una copia de cierta variable entre todos los objetos de una clase. En esos casos se utiliza un *campo* `static` (al cual se le conoce como una *variable de clase*), entre otros. Una variable de clase representa *información en toda la clase* (todos los objetos de la clase comparten la misma pieza de datos). La declaración de un miembro `static` comienza con la palabra clave `static`.

Determinaremos la necesidad de datos `static` que puedan usarse en toda la clase con un videojuego como ejemplo. Suponga que tenemos un videojuego con `Marcianos` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y deseoso de atacar a otras criaturas espaciales cuando sabe que hay al menos otros cinco `Marcianos` presentes. Si están presentes menos de cinco `Marcianos`, cada `Marciano` se vuelve cobarde, por lo que cada uno de ellos necesita saber el valor de `cuentaMarcianos`. Podríamos dotar a la clase `Marciano` con la variable `cuentaMarcianos` como datos de instancia, y cada vez que creáramos un nuevo `Marciano`, tendríamos que actualizar la variable de instancia `cuentaMarcianos` en todos los objetos `Marciano`. Las copias redundantes desperdician espacio y tiempo en actualizar cada una de las copias de la variable, además de ser un proceso propenso a errores. En vez de ello, declaramos a `cuentaMarcianos` como `static`. Esto hace que `cuentaMarcianos` esté disponible en toda la clase. Cada objeto `Marciano` puede ver la `cuentaMarcianos` como si fueran datos de instancia de la clase `Marciano`, pero Java sólo mantiene una copia de la variable `cuentaMarcianos` estática. Esto nos ahorra espacio. Ahorramos tiempo al hacer que el constructor de `Marciano` incremente a la variable `cuentaMarcianos` estática. Y como sólo hay una copia, no tenemos que incrementar cada una de las copias de `cuentaMarcianos` para cada uno de los objetos `Marciano`.



Observación de ingeniería de software 8.14

Use una variable de clase cuando todos los objetos de una clase tengan que utilizar la misma copia de la variable.

Aunque las variables de clase pueden parecer variables globales, tienen alcance a nivel de clase. Los miembros `static` de una clase `public` pueden utilizarse a través de una referencia a cualquier objeto de esa clase, o calificando el nombre del miembro con el nombre de la clase y un punto (`.`), como en `Math.random()`. Los miembros de clase `private static` de una clase pueden utilizarse solamente a través de los métodos de esa clase. En realidad, los miembros de clase `static` existen a pesar de que no existan objetos de esa clase; están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución. Para acceder a un miembro `public static` cuando no existen objetos de la clase, se debe anteponer el nombre de la clase y un punto (`.`) al miembro de la clase. Para acceder a un miembro `private static` cuando no existen objetos de la clase debe proporcionarse un método `public static`, y para llamar a este método se debe calificar su nombre con el nombre de la clase y un punto.



Observación de ingeniería de software 8.15

Las variables de clase y los métodos estáticos existen, y pueden utilizarse, incluso aunque no se hayan instanciado objetos de esa clase.

En nuestro siguiente programa declaramos dos clases: `Empleado` (figura 8.12) y `PruebaEmpleado` (figura 8.13). La clase `Empleado` declara a la variable `private static` llamada `cuenta` (figura 8.12, línea 6), y al método `public static` llamado `obtenerCuenta` (figura 8.12, líneas 43 a 46). La variable de clase `cuenta` se inicializa con cero en la línea 6. Si no se inicializa una variable de clase `static`, el compilador asigna a esa variable su valor predeterminado. La variable de clase `cuenta` mantiene la cuenta del número de objetos de la clase `Empleado` que residen actualmente en memoria. Esto incluye a los objetos que ya hayan sido marcados para la recolección de basura, pero que no se han reclamado todavía.

http://librosysolucionarios.net


```
1  // Fig. 8.12: Empleado.java
2  // Declaración de la clase Empleado.
3  public class Empleado {
4     private String primerNombre;
5     private String apellidoPaterno;
6     private static int cuenta = 0; // número de objetos en la memoria
7
8     // inicializar empleado, sumar 1 a la cuenta estática y
9     // mostrar String indicando que se llamó al constructor
10    public Empleado( String nombre, String apellido )
11    {
12       primerNombre = nombre;
13       apellidoPaterno = apellido;
14
15       ++cuenta;  // incrementar cuenta estática de empleados
16       System.out.println( "Constructor de Empleado: " +
17          primerNombre + " " + apellidoPaterno );
18    }
19
20    // restar 1 de la cuenta estática cuando el recolector de basura
21    // llama a finalize para limpiar el objeto y mostrar String
22    // indicando que se llamó a finalize
23    protected void finalize()
24    {
25       --cuenta;  // decrementar cuenta estática de empleados
26       System.out.println( "Finalizador de Empleado: " +
27          primerNombre + " " + apellidoPaterno + "; cuenta = " + cuenta );
28    }
29
30    // obtener primer nombre
31    public String obtenerPrimerNombre()
32    {
33       return primerNombre;
34    }
35
36    // obtener apellido
37    public String obtenerApellidoPaterno()
38    {
39       return apellidoPaterno;
40    }
41
42    // método estático para obtener el valor de la cuenta estática
43    public static int obtenerCuenta()
44    {
45       return cuenta;
46    }
47
48 } // fin de la clase Empleado
```

Figura 8.12 Variable `static` utilizada para mantener una cuenta del número de objetos `Empleado` en memoria.

Cuando existen objetos `Empleado`, el miembro `cuenta` puede utilizarse en cualquier método de cualquier objeto `Empleado`; en este ejemplo incrementamos a `cuenta` en el constructor (línea 15) y lo decrementamos en el finalizador (línea 25). Cuando no existen objetos de la clase `Empleado`, el miembro `cuenta` puede aún referenciarse, pero sólo a través de una llamada al método `public static` llamado `obtener-`

`Cuenta` (líneas 43 a 46), como en `Empleado.obtenerCuenta()` que devuelve el número de objetos `Empleado` actualmente en memoria. Sin embargo, cuando hay objetos instanciados el método `obtenerCuenta` puede llamarse también mediante una referencia a uno de los objetos, como en la llamada `e1.obtenerCuenta()`.

Buena práctica de programación 8.6

Siempre debe invocar a los métodos `static` utilizando el nombre de la clase y un punto (`.`). Esto enfatiza, a los demás programadores que leen su código, que el método que se está llamando es un método `static`.

Observe que la clase `Empleado` tiene un método `finalize` (líneas 23 a 28). Este método se incluye solamente para mostrar cuándo se ejecuta el recolector de basura en este programa. El método `finalize` se declara normalmente como `protected`, de manera que no forma parte de los servicios `public` de una clase. Hablaremos sobre el modificador de acceso de miembros `protected` detalladamente en el capítulo 9.

El método `main` de `PruebaEmpleado` (figura 8.13) crea instancias de dos objetos `Empleado` (líneas 15 y 16). Cuando se invoca el constructor de cada objeto `Empleado`, en las líneas 12 y 13 de la figura 8.12 se almacenan referencias a los objetos `String` del primer nombre y apellido paterno de ese objeto `Empleado`.[6]

Cuando `main` ha terminado de usar los dos objetos `Empleado`, las referencias `e1` y `e2` se establecen en `null`, en las líneas 32 y 33. En este punto, las referencias `e1` y `e2` ya no hacen referencia a los objetos que se instanciaron en las líneas 15 y 16. Esto "marca a los objetos para la recolección de basura", ya que no existen más referencias a esos objetos en el programa.

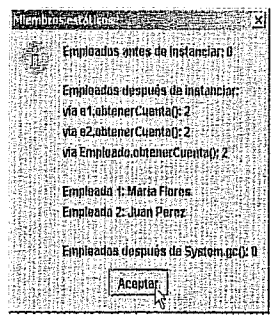
```
1  // Fig. 8.13: PruebaEmpleado.java
2  // Prueba de la clase Empleado con la variable de clase estática,
3  // el método de clase estático y memoria dinámica.
4  import javax.swing.*;
5
6  public class PruebaEmpleado {
7
8     public static void main( String args[] )
9     {
10       // demostrar que cuenta es 0 antes de crear Empleados
11       String salida = "Empleados antes de instanciar: " +
12          Empleado.obtenerCuenta();
13
14       // crear dos Empleados; la cuenta debe ser 2
15       Empleado e1 = new Empleado( "Maria", "Flores" );
16       Empleado e2 = new Empleado( "Juan", "Perez" );
17
18       // demostrar que la cuenta es 2 después de crear dos Empleados
19       salida += "\n\nEmpleados después de instanciar: " +
20          "\nvia e1.obtenerCuenta(): " + e1.obtenerCuenta() +
21          "\nvia e2.obtenerCuenta(): " + e2.obtenerCuenta() +
22          "\nvia Empleado.obtenerCuenta(): " + Empleado.obtenerCuenta();
23
```

Figura 8.13 Demostración de un miembro `static`. (Parte 1 de 2.)*

6. Estas dos instrucciones *no* sacan copias de los argumentos `String` originales. En realidad, los objetos `String` en Java son inmutables (no pueden modificarse una vez que son creados). Como no puede utilizarse una referencia para modificar a un objeto `String`, es seguro tener muchas referencias a un solo objeto `String`. Éste no es normalmente el caso para la mayoría de las otras clases en Java. Si los objetos `String` son inmutables, ¿por qué podemos utilizar los operadores `+` y `+=` para concatenar objetos `String`? Como veremos en el capítulo 11, las operaciones de concatenación de objetos `String` en realidad producen un nuevo objeto `String`, el cual contiene los valores concatenados. Los objetos `String` originales no se modifican.

*Algunos programas como el de la figura 8.13 muestran una parte de la salida en ventanas de windows y otra parte en la ventana de Ms-Dos en la que se ejecuta el programa. Como Java no ofrece soporte para caracteres especiales (como acentos) en la consola predeterminada, optamos por omitir los acentos cuando la salida del programa se muestre en ventanas de Ms-Dos. (Nota del traductor.)


```
24       // obtener nombres de Empleados
25       salida += "\n\nEmpleado 1: " + e1.obtenerPrimerNombre() +
26          " " + e1.obtenerApellidoPaterno() + "\nEmpleado 2: " +
27          e2.obtenerPrimerNombre() + " " + e2.obtenerApellidoPaterno();
28
29       // decrementar la cuenta de referencia para cada objeto Empleado; en este
30       // ejemplo, hay sólo una referencia a cada Empleado, por lo que estas
31       // instrucciones marcan a cada objeto Empleado para la recolección de basura
32       e1 = null;
33       e2 = null;
34
35       System.gc(); // sugerir una llamada al recolector de basura
36
37       // mostrar cuenta de objetos Empleado después de llamar al recolector
            de basura; la cuenta
38       // mostrada puede ser 0, 1 o 2 dependiendo de si el recolector de basura
39       // se ejecuta inmediatamente o no y del número de objetos Empleado
            recolectados
40       salida += "\n\nEmpleados después de System.gc(): " +
41          Empleado.obtenerCuenta();
42
43       JOptionPane.showMessageDialog( null, salida,
44          "Miembros estáticos", JOptionPane.INFORMATION_MESSAGE );
45
46       System.exit( 0 );
47    }
48
49 } // fin de la clase PruebaEmpleado
```




```
Constructor de Empleado: Maria Flores
Constructor de Empleado: Juan Perez
Finalizador de Empleado: Maria Flores; cuenta = 1
Finalizador de Empleado: Juan Perez; cuenta = 0
```

Figura 8.13 Demostración de un miembro `static`. (Parte 2 de 2.)

De un momento a otro, el recolector de basura podría reclamar la memoria para estos objetos (o el sistema operativo reclama la memoria cuando el programa termina). No se garantiza cuándo se va a ejecutar el recolector de basura (o si acaso se va a ejecutar), por lo que este programa hace una llamada explícita al recolector de basura en la línea 35 utilizando el método `static` llamado `gc`, de la clase `System` (paquete `java.lang`) para indicar que el recolector de basura debe realizar su mejor esfuerzo por tratar de reclamar objetos que sean

http://librosysolucionarios.net

elegibles para la recolección de basura. Sin embargo, esto es sólo el mejor esfuerzo; es posible que no se recolecten objetos, o que se recolecte un subconjunto de los objetos en la basura. En nuestro ejemplo, el recolector de basura se ejecutó antes de que en las líneas 43 y 44 se mostraran los resultados del programa. La última línea de la salida indica que el número de objetos `Empleado` en memoria es 0 después de la llamada a `System.gc()`. Además, las últimas dos líneas de la salida en la ventana de comandos muestran que el objeto `Empleado` para `Maria Flores` se finalizó antes que el objeto `Empleado` para `Juan Perez`. Recuerde, no se garantiza que el recolector de basura se ejecute al invocar a `System.gc()`, ni se garantiza que se recolecten los objetos en un orden específico, por lo que es posible que la salida de este programa pueda ser diferente en su sistema.

(*Nota:* Un método declarado como `static` no puede tener acceso a los miembros no `static` de una clase. A diferencia de los métodos no `static`, un método `static` no tiene referencia `this` ya que las variables y los métodos `static` existen en forma independiente de cualquier objeto de una clase, sin importar que se hayan instanciado o no objetos de esa clase.)

Error común de programación 8.9

Es un error de sintaxis para un método `static` llamar a un método de instancia de manera directa o acceder a una variable de instancia directamente. Sin embargo, un método `static` puede llamar a un método de instancia o acceder a una variable de instancia a través de una referencia a un objeto, si dicha referencia está disponible en el método.

Error común de programación 8.10

Hacer referencia a la referencia `this` en un método `static` es un error de sintaxis; no hay objeto de la clase al cual hacer referencia.

8.12 Variables de instancia finales

El *principio del menor privilegio* es uno de los principios más fundamentales de la buena ingeniería de software. Veamos ahora una manera en la que este principio se aplica a las variables de instancia.

Algunas variables de instancia necesitan modificarse, mientras que otras no. El programador puede utilizar la palabra clave `final` para especificar que una variable no puede modificarse (es decir, que sea constante) y que cualquier intento por modificar esa variable es un error. Por ejemplo,

```
private final int INCREMENTO = 5;
```

declara una variable de instancia `final` (constante) llamada `INCREMENTO`, de tipo `int`, y la inicializa en 5. No se requiere que las constantes se inicialicen al momento de declararse. También pueden inicializarse mediante cada uno de los constructores de la clase.

Observación de ingeniería de software 8.16

Declarar una variable de instancia como `final` ayuda a hacer valer el principio del menor privilegio. Si una variable de instancia no debe modificarse, declárela como `final` para evitar su modificación.

Error común de programación 8.11

Intentar modificar una variable de instancia `final` después de inicializarla es un error de compilación.

Tip para prevenir errores 8.3

Los intentos accidentales por modificar una variable de instancia `final` se atrapan en tiempo de compilación, en vez de que produzcan errores en tiempo de ejecución. Siempre es preferible obtener errores en tiempo de compilación, si es posible, en vez de permitir que los errores se escapen hasta el tiempo de ejecución (en donde los estudios han encontrado que el costo de reparación es comúnmente hasta diez veces más alto).

En la figura 8.14 se muestran dos clases: la clase de applet `PruebaIncremento` (líneas 7 a 30) y la clase `Incremento` (líneas 33 a 57). Esta clase contiene una variable de instancia `final` llamada `INCREMENTO`, de tipo `int` (línea 36). La variable final no se inicializa en su declaración, por lo que debe inicializarse mediante un constructor de la clase. El constructor de las líneas 39 a 42 recibe el parámetro `int` de nombre `valorIncremento`, el cual se asigna a `INCREMENTO` en la línea 41. Una variable `final` no puede modificarse mediante una asignación, una vez que se inicializa. En el applet `PruebaIncremento` se crea un objeto de la

clase `Incremento` en la línea 14, y se proporciona el valor 5 como argumento para el constructor, el cual se asigna a la constante `INCREMENTO`.

Error común de programación 8.12

El no inicializar una variable de instancia `final` en su declaración o en cada constructor de la clase es un error de sintaxis.

```
1  // Fig. 8.14: PruebaIncremento.java
2  // Inicialización de una variable final.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaIncremento extends JApplet implements ActionListener {
8     private Incremento objetoIncremento;
9     private JButton boton;
10
11    // configurar GUI
12    public void init()
13    {
14       objetoIncremento = new Incremento( 5 );
15
16       Container contenedor = getContentPane();
17
18       boton = new JButton( "Haga clic para incrementar" );
19       boton.addActionListener( this );
20       contenedor.add( boton );
21    }
22
23    // sumar INCREMENTO al total cuando el usuario haga clic en el botón
24    public void actionPerformed( ActionEvent eventoAccion )
25    {
26       objetoIncremento.incrementar();
27       showStatus( objetoIncremento.aStringIncremento() );
28    }
29
30 } // fin de la clase PruebaIncremento
31
32 // clase que contiene variable constante
33 class Incremento {
34    private int cuenta = 0;          // número de incrementos
35    private int total = 0;           // total de todos los incrementos
36    private final int INCREMENTO; // variable constante
37
38    // inicializar constante INCREMENTO
39    public Incremento( int valorIncremento )
40    {
41       INCREMENTO = valorIncremento; // inicializar variable constante (una vez)
42    }
43
44    // sumar INCREMENTO al total y sumar 1 a cuenta
45    public void incrementar()
46    {
47       total += INCREMENTO;
48       ++cuenta;
```

Figura 8.14 Variable `final` inicializada con argumento del constructor. (Parte 1 de 2.)

http://librosysolucionarios.net

```
49     }
50
51     // devolver representación String de los datos de un objeto Incremento
52     public String aStringIncremento()
53     {
54        return "Después de incremento " + cuenta + ": total = " + total;
55     }
56
57  } // fin de la clase PruebaIncremento
```

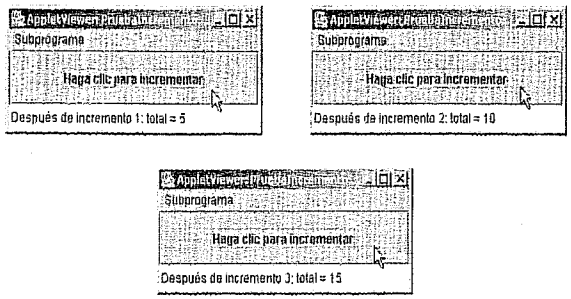


Figura 8.14 Variable `final` inicializada con argumento del constructor. (Parte 2 de 2.)

```
PruebaIncremento.java:40: variable INCREMENTO might not have been initialized
      {
       ^
1 error
```

Figura 8.15 La variable `final` `INCREMENTO` debe inicializarse.

Si una variable final no se inicializa, se produce un error de compilación. Para demostrar esto, colocamos la línea 41 de la figura 8.14 en un comentario y volvimos a compilar el programa. En la figura 8.15 se muestra el mensaje de error producido por el compilador en este caso.

8.13 Creación de paquetes

Hemos visto en casi todos los ejemplos de este libro que las clases e interfaces[7] de bibliotecas preexistentes, como la API de Java, pueden importarse en un programa en Java. Cada clase e interfaz en la API de Java pertenece a un paquete que contiene un grupo de clases e interfaces relacionadas. A medida que las aplicaciones se vuelven más complejas, los paquetes ayudan a los programadores a manejar la complejidad de los componentes de las aplicaciones. Los paquetes también facilitan la reutilización de software al permitir que los programas importen clases de otros paquetes (como lo hemos hecho en la mayoría de los ejemplos). Otro beneficio de los paquetes es que proporcionan una convención para *nombres de clase únicos*. Con cientos de miles de programadores en Java en todo el mundo, hay una muy buena probabilidad de que los nombres que usted elija para sus clases tengan conflictos con los nombres que otros programadores elijan para las suyas. En esta sección lo introduciremos al proceso de crear sus propios paquetes.

En la aplicación de las figuras 8.16 y 8.17 le mostraremos cómo crear su propio paquete y utilizar una clase de ese paquete en un programa. Los pasos para crear una clase reutilizable son:

7. En la sección 6.8 presentamos las interfaces por primera vez, y las describiremos con detalle en el capítulo 10.

1. Declare una clase como `public`. Si la clase no es `public`, sólo podrá ser utilizada por otras clases dentro del mismo paquete.
2. Elija el nombre del paquete y agregue una *declaración* **`package`** al archivo de código fuente para la declaración de la clase reutilizable. Sólo puede haber una declaración `package` en un archivo de código fuente en Java, y ésta debe ir antes de cualquier otra declaración o instrucción en el archivo.
3. Compile la clase, de manera que se coloque en la estructura de directorio del paquete apropiado.
4. Importe la clase reutilizable en un programa y utilícela.



Error común de programación 8.13

Se produce un error de sintaxis si cualquier declaración o instrucción aparece en un archivo de código fuente en Java antes de la declaración **`package`** *(si es que hay una) en el archivo.*

Optamos por mostrar el *paso 1* modificando la clase `public` llamada `Tiempo1`, la cual declaramos en la figura 8.1. La nueva versión se muestra en la figura 8.16. No se ha realizado ninguna modificación a la implementación de la clase, por lo que aquí no hablaremos de nuevo sobre sus detalles de implementación.

Para cumplir con el *paso 2*, agregamos una declaración `package` al principio del archivo. En la línea 3 se utiliza una declaración `package` para declarar un paquete llamado `com.deitel.cpej5.cap08`. Al colocar una declaración `package` al principio de un archivo de código fuente en Java, se indica que la clase (o clases) declarada(s) en el archivo forma(n) parte del paquete especificado. Sólo las declaraciones `package`, las declaraciones `import` y los comentarios pueden aparecer fuera de las llaves de la declaración de una clase. Un archivo de código fuente en Java tiene el siguiente orden: primero una declaración `package` (si la hay), luego las declaraciones `import` (si las hay), y después las declaraciones de clase. Sólo una de las declaraciones de clase en un archivo puede ser pública. Las demás clases en el archivo se colocan también en el paquete, pero son reutilizables desde cualquier otra clase en ese paquete; no pueden importarse en clases de otros paquetes. Se encuentran en el paquete para dar soporte a la clase reutilizable del archivo.

```
1  // Fig. 8.16: Tiempo1.java
2  // Declaración de la clase Tiempo1 que mantiene la hora en formato de 24 horas.
3  package com.deitel.cpej5.cap08;
4
5  import java.text.DecimalFormat;
6
7  public class Tiempo1 extends Object {
8     private int hora;       // 0 - 23
9     private int minuto;     // 0 - 59
10    private int segundo;    // 0 - 59
11
12    // El constructor de Tiempo1 inicializa cada variable de instancia en cero;
13    // se asegura de que cada objeto Tiempo1 inicie en un estado consistente
14    public Tiempo1()
15    {
16       establecerHora( 0, 0, 0 );
17    }
18
19    // establecer un nuevo valor de hora utilizando hora universal; realizar
20    // comprobaciones de validez en los datos; establecer valores inválidos en cero
21    public void establecerHora( int h, int m, int s )
22    {
23       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
24       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
25       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
```

Figura 8.16 Empaquetamiento de la clase `Tiempo1` para su reutilización. (Parte 1 de 2.)

http://librosysolucionarios.net

```
26    }
27
28    // convertir a String en formato de hora universal
29    public String aStringUniversal()
30    {
31       DecimalFormat dosDigitos = new DecimalFormat( "00" );
32
33       return dosDigitos.format( hora ) + ":" +
34          dosDigitos.format( minuto ) + ":" + dosDigitos.format( segundo );
35    }
36
37    // convertir a String en formato de hora estándar
38    public String aStringEstandar()
39    {
40       DecimalFormat dosDigitos = new DecimalFormat( "00" );
41
42       return ( (hora == 12 || hora == 0) ? 12 : hora % 12 ) + ":" +
43          dosDigitos.format( minuto ) + ":" + dosDigitos.format( segundo ) +
44          ( hora < 12 ? " AM" : " PM" );
45    }
46
47 } // fin de la clase Tiempo1
```

Figura 8.16 Empaquetamiento de la clase **Tiempo1** para su reutilización. (Parte 2 de 2.)

En un esfuerzo por proporcionar nombres únicos para cada paquete, Sun Microsystems especifica una convención para nombrar paquetes, que deben seguir todos los programadores en Java. Todos los nombres de paquetes que usted vaya a crear deben empezar con su nombre de dominio de Internet en orden inverso. Por ejemplo, para el nombre de dominio de Internet **deitel.com**, el nombre del paquete debe comenzar con **com.deitel**. Para el nombre de dominio *suescuela*.**edu**, el nombre del paquete debe empezar con **edu**.*suescuela*. Una vez invertido el nombre de dominio, puede elegir cualquier otro nombre que desee para su paquete. Si usted forma parte de una compañía con muchas divisiones, o de una universidad con muchos planteles, tal vez quiera utilizar el nombre de su división o plantel como el siguiente nombre en el paquete. Optamos por utilizar **cpej5** como el siguiente nombre en el nombre de nuestro paquete, para indicar que esta clase es de *Cómo programar en Java, quinta edición*. La última parte en el nombre de nuestro paquete especifica que este paquete es para el capítulo 8 (**cap08**).[8]

El *paso 3* es compilar la clase para guardarla en el paquete apropiado. Cuando se compila un archivo en Java que contiene una declaración **package**, el archivo de clase resultante se coloca en la estructura de directorio especificada por la declaración **package**. La declaración **package** de la figura 8.16 indica que la clase **Tiempo1** debe colocarse en el directorio **cap08**. Los otros nombres (**com**, **deitel** y **cpej5**) son también directorios. Los nombres de directorio en la declaración **package** especifican la ubicación exacta de las clases en el paquete. Si estos directorios no existen antes de compilar la clase, el compilador puede crearlos.

Al compilar una clase en un paquete, la opción de línea de comandos (*-d*) hace que el compilador **javac** cree los directorios apropiados, con base en la declaración **package** de la clase. Esta opción también especifica en dónde crear (o ubicar) los directorios. Por ejemplo, en una ventana de comandos, utilizamos el siguiente comando de compilación:

```
javac -d . Tiempo1.java
```

para especificar que el primer directorio en el nombre de nuestro paquete debe colocarse en el directorio actual. El **.** después de **-d** en el comando anterior representa al directorio actual en los sistemas operativos Windows, UNIX y Linux (y en varios otros también). Después de ejecutar el comando de compilación, el directorio actual contiene un directorio llamado **com**, **com** contiene un directorio llamado **deitel**, **deitel** contiene un directorio llamado **cpej5** y **cpej5** contiene un directorio llamado **cap08**. En este directorio puede encontrar el archivo **Tiempo1.class**. (*Nota:* Si no utiliza la opción **-d**, entonces deberá copiar o mover el archivo de clase al directorio del paquete apropiado, después de compilarlo.)

8. Utilizamos nuestros propios paquetes varias veces a lo largo de este libro. Usted puede determinar el capítulo en el que se declara cada una de nuestras clases reutilizables con sólo ver la última parte del nombre del paquete en la instrucción **import**. Ésta aparece antes del nombre de la clase que se va a importar, o antes del * si no se especifica una clase en particular.

El nombre del paquete es parte del nombre de la clase. En este ejemplo, el nombre de la clase es realmente **com.deitel.cpej5.cap08.Tiempo1**. Puede utilizar este nombre *completamente calificado* en sus programas, o puede importar la clase (mediante **import**) y utilizar su nombre simple (**Tiempo1**) en el programa. Si otro paquete también contiene una clase **Tiempo1**, pueden utilizarse los nombres de clase completamente calificados para diferenciar entre las clases en el programa y así evitar un *conflicto de nombres* (lo que también se conoce como una *colisión de nombres*).

Una vez compilada la clase y almacenada en su paquete, ésta puede importarse en los programas (*paso 4*). En la línea 8 de la aplicación **PruebaTiempo1** de la figura 8.17, se especifica que se debe importar la clase **Tiempo1** para utilizarla en la clase **PruebaTiempo1**. (*Nota:* Las clases en un paquete no necesitan importar a otras clases del mismo paquete.)

```
1  // Fig. 8.17: PruebaTiempo1.java
2  // Clase PruebaTiempo1 que utiliza la clase Tiempo1.
3
4  // paquetes de Java
5  import javax.swing.JOptionPane;
6
7  // paquetes de Deitel
8  import com.deitel.cpej5.cap08.Tiempo1; // importar la clase Tiempo1
9
10 public class PruebaTiempo1 {
11
12    public static void main( String args[] )
13    {
14       Tiempo1 tiempo = new Tiempo1(); // llamar al constructor de Tiempo1
15
16       // anexar versión String de tiempo a salida String
17       String salida = "La hora universal inicial es: " +
18          tiempo.aStringUniversal() + "\nLa hora estándar inicial es: " +
19          tiempo.aStringEstandar();
20
21       // cambiar tiempo y anexar hora actualizada a salida
22       tiempo.establecerHora( 13, 27, 6 );
23       salida += "\n\nLa hora universal después de establecerHora es: " +
24          tiempo.aStringUniversal() +
25          "\nLa hora estándar después de establecerHora es: " +
26            tiempo.aStringEstandar();
27
28       // establecer tiempo con valores inválidos; anexar hora actualizada
29          a salida
30       tiempo.establecerHora( 99, 99, 99 );
31       salida += "\n\nDespués de intentar ajustes inválidos: " +
32          "\nHora universal: " + tiempo.aStringUniversal() +
33          "\nHora estándar: " + tiempo.aStringEstandar();
34
35       JOptionPane.showMessageDialog( null, salida,
          "Prueba de la clase Tiempo1", JOptionPane.INFORMATION_MESSAGE );
```

Figura 8.17 Cómo importar su propia clase desde un paquete. (Parte 1 de 2.)

http://librosysolucionarios.net

```
36         System.exit( 0 );
37
38      } // fin de main
39
40   } // fin de la clase PruebaTiempo1
```

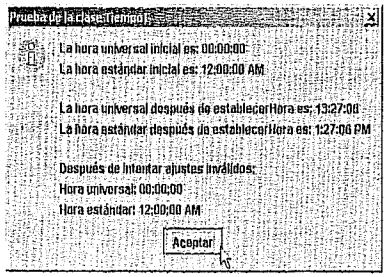


Figura 8.17 Cómo importar su propia clase desde un paquete. (Parte 2 de 2.)

Al compilar `PruebaTiempo1`, `javac` debe localizar el archivo `.class` de `Tiempo1` para asegurarse de que la clase `PruebaTiempo1` utilice a la clase `Tiempo1` correctamente. El compilador utiliza un objeto especial llamado *cargador de clases* para localizar las clases que necesita. El cargador de clases empieza buscando en las clases estándar de Java que se incluyen con el J2SDK. Después busca los *paquetes opcionales*. Java 2 cuenta con un *mecanismo de extensión* que permite que se agreguen nuevos paquetes (opcionales) a Java para fines de desarrollo y ejecución.[9] Si la clase no se encuentra en las clases estándar de Java ni en las clases de extensión, el cargador de clases busca en la *ruta de clases*, la cual contiene una lista de ubicaciones en las que se almacenan las clases. La ruta de clases consiste en una lista de directorios o *archivos de ficheros*, cada uno separado por un *separador de directorio*: un signo de punto y coma (`;`) en Windows o un signo de dos puntos (`:`) en UNIX/Linux/Mac OS X. Los archivos de ficheros son archivos individuales que contienen directorios de otros archivos, generalmente en formato comprimido. Por ejemplo, las clases estándar de Java están contenidas en el archivo de ficheros `rt.jar`, el cual se instala junto con el J2SDK. Los archivos de ficheros generalmente terminan con la extensión `.jar` o `.zip`. Los directorios y archivos de ficheros que se especifican en la ruta de clases contienen las clases que usted desea poner a disponibilidad del compilador y la máquina virtual de Java.

De manera predeterminada, la ruta de clases consiste sólo del directorio actual. Sin embargo, la ruta de clases puede modificarse[10] de la siguiente manera:

1. proporcionando la opción `-classpath` al compilador `javac` o
2. estableciendo la variable de entorno `CLASSPATH` (una variable especial que usted define y el sistema operativo mantiene, de manera que las aplicaciones puedan buscar clases en las ubicaciones especificadas).

Error común de programación 8.14

Al especificar una ruta de clases explícita se elimina el directorio actual de la ruta de clases. Esto evita que las clases en el directorio actual (incluyendo los paquetes en ese directorio) se carguen correctamente. Si deben cargarse clases del directorio actual, incluya a este directorio (`.`) en la ruta de clases explícita.

9. El mecanismo de extensión se encuentra más allá del alcance de este libro. Para obtener más información, visite la página `java.sun.com/j2se/1.4.1/docs/guide/extensions`.
10. Para obtener más información sobre la ruta de clases, visite la página `java.sun.com/j2se/1.4.1/docs/tooldocs/windows/classpath.html` para Windows, o `java.sun.com/j2se/1.4.1/docs/tooldocs/solaris/classpath.html` para Solaris/Linux.

Observación de ingeniería de software 8.17

En general, es una mejor práctica utilizar la opción `-classpath` del compilador en vez de usar la variable de entorno `CLASSPATH` para especificar la ruta de clases para un programa. De esta manera, cada aplicación puede tener su propia ruta de clases.

Tip para prevenir errores 8.4

Al especificar la ruta de clases con la variable de entorno `CLASSPATH` se pueden producir errores sutiles y difíciles de localizar en los programas que utilicen versiones distintas del mismo paquete.

Para el ejemplo de las figuras 8.16 y 8.17, no especificamos una ruta de clases explícita. Por lo tanto, para localizar las clases en el paquete `com.deitel.cpej5.cap08` de este ejemplo, el cargador de clases busca en el directorio actual el primer nombre en el paquete: `com`. A continuación, el cargador de clases navega por la estructura de directorios. El directorio `com` contiene al subdirectorio `deitel`. Este subdirectorio contiene al subdirectorio `cpej5`. Finalmente, el directorio `cpej5` contiene al subdirectorio `cap08`. En este subdirectorio se encuentra el archivo `Tiempo1.class`, que se carga mediante el cargador de clases para asegurar que la clase se utilice apropiadamente en nuestro programa.

El proceso de localizar las clases para ejecutar el programa es similar a localizar las clases para compilarlo. Al igual que el compilador, el intérprete `java` utiliza un cargador de clases que busca primero en las clases estándar y de extensión, y después busca en la ruta de clases (el directorio actual, de manera predeterminada). La ruta de clases para el intérprete puede especificarse en forma explícita, utilizando cualquiera de las técnicas descritas para el compilador. Al igual que con el compilador, es mejor especificar al intérprete una ruta de clases individual para cada programa, mediante las opciones de la línea de comandos. Usted puede especificar la ruta de clases para el intérprete `java` mediante las opciones de línea de comandos `-classpath` o `-cp`, seguidas de una lista de directorios o archivos de ficheros separados por signos de punto y coma (`;`) en Windows, o signos de dos puntos (`:`) en UNIX/Linux/Mac OS X.

8.14 Acceso a los paquetes

Si no se especifica un modificador de acceso (`public`, `protected` o `private`) para un método o variable al declararse en una clase, se considerará que el método o variable tiene *acceso a nivel de paquete*. En un programa que consiste de una declaración de clase, esto no tiene un efecto específico sobre el programa. No obstante, si un programa utiliza varias clases del mismo paquete (es decir, un grupo de clases relacionadas), estas clases pueden acceder a los métodos y campos (con acceso a nivel de paquete) de cada una de las otras clases directamente, a través de referencias a objetos de las clases apropiadas.

Analicemos un ejemplo mecánico del acceso a nivel de paquete. La aplicación de la figura 8.18 contiene dos clases: la clase de aplicación `PruebaDatosPaquete` (líneas 6 a 30) y la clase `DatosPaquete` (líneas 33 a 50). En las líneas 34 y 35 de la declaración de la clase `DatosPaquete`, se declaran las variables de instancia `numero` y `cadena` sin modificadores de acceso; por lo tanto, se consideran como variables de instancia con acceso a nivel de paquete. En el método `main` de la aplicación `PruebaDatosPaquete` se crea una instancia de la clase `DatosPaquete` (línea 10), para mostrar la habilidad de modificar las variables de instancia de `DatosPaquete` directamente (como se muestra en las líneas 17 y 18). Los resultados de la modificación pueden verse en la ventana de salida.

```
1  // Fig. 8.18: PruebaDatosPaquete.java
2  // Las clases en el mismo paquete (es decir, el mismo directorio) pueden
3  // utilizar los datos con acceso de paquete, de otras clases en el mismo paquete.
4  import javax.swing.JOptionPane;
5
6  public class PruebaDatosPaquete {
7
```

Figura 8.18 Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete. (Parte 1 de 2.)

http://librosysolucionarios.net

```
8     public static void main( String args[] )
9     {
10       DatosPaquete datosPaquete = new DatosPaquete();
11
12       // anexar representación String de datosPaquete a salida
13       String salida = "Después de instanciar:\n" +
14          datosPaquete.aStringDatosPaquete();
15
16       // cambiar los datos con acceso de paquete en el objeto datosPaquete
17       datosPaquete.numero = 77;
18       datosPaquete.cadena = "Adiós";
19
20       // anexar la representación String de datosPaquete a salida
21       salida += "\nDespués de cambiar valores:\n" +
22          datosPaquete.aStringDatosPaquete();
23
24       JOptionPane.showMessageDialog( null, salida, "Acceso a nivel de paquete",
25          JOptionPane.INFORMATION_MESSAGE );
26
27       System.exit( 0 );
28    }
29
30 } // fin de la clase PruebaDatosPaquete
31
32 // la clase con variables de instancia con acceso de paquete
33 class DatosPaquete {
34    int numero;    // variable de instancia con acceso de paquete
35    String cadena;  // variable de instancia con acceso de paquete
36
37    // constructor
38    public DatosPaquete()
39    {
40       numero = 0;
41       cadena = "Hola";
42    }
43
44    // devolver representación String del objeto DatosPaquete
45    public String aStringDatosPaquete()
46    {
47       return "numero: " + numero + "     cadena: " + cadena;
48    }
49
50 } // fin de la clase DatosPaquete
```

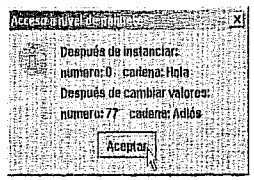


Figura 8.18 Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete. (Parte 2 de 2.)

Al compilar este programa, el compilador produce dos archivos separados: un archivo de clase para la clase `PruebaDatosPaquete` y un archivo de clase para la clase `DatosPaquete`. Recuerde que en de la sec-

http://librosysolucionarios.net

ción 8.5 vimos que toda clase de Java tiene su propio archivo de clase con la extensión `.class`. El compilador coloca las dos clases de la figura 8.18 en el mismo directorio, por lo que se consideran parte del mismo paquete. (Ciertamente, están relacionadas por el hecho de que se encuentran en el mismo archivo `.java`.) Como forman parte del mismo paquete, la clase `PruebaDatosPaquete` tiene permiso para modificar los datos con acceso a nivel de paquete, de los objetos de la clase `DatosPaquete`.

Observación de ingeniería de software 8.18



Algunas personas en la comunidad de POO sienten que el acceso a nivel de paquete corrompe el ocultamiento de la información, y debilita el valor de la metodología de diseño orientado a objetos, ya que el programador debe asumir la responsabilidad de comprobar los errores y validar los datos en cualquier código que manipule a los miembros de datos con acceso a nivel de paquete.

8.15 Reutilización de software

Los programadores en Java se concentran en fabricar nuevas clases y reutilizar las clases existentes. Existen muchas *bibliotecas de clases*, y se están desarrollando más a nivel mundial. El software se construye entonces a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portables y ampliamente utilizados. Este tipo de reutilización de software agiliza el desarrollo de software poderoso de alta calidad. El *desarrollo rápido de aplicaciones (RAD)* es de gran interés hoy en día.

Los programadores en Java tienen ahora miles de clases a escoger en la API de Java, para ayudarse a implementar programas. Evidentemente, Java no es sólo un lenguaje de programación. Es un marco de trabajo en el que los desarrolladores en Java pueden trabajar para lograr una verdadera reutilización y un desarrollo rápido de aplicaciones. Los programadores en Java pueden enfocarse en la tarea principal al desarrollar sus programas, y dejan los detalles de nivel inferior a las clases de la API de Java. Por ejemplo, para escribir un programa que dibuja gráficos, un programador en Java no requiere de un conocimiento sobre los gráficos en todas las plataformas computacionales en las que el programa vaya a ejecutarse. En vez de ello, un programador en Java puede concentrarse en aprender las herramientas para gráficos en Java (que son bastante substanciales, y cada día aumentan más) y escribir un programa en Java que dibuje los gráficos, utilizando clases de la API de Java como `Graphics`. Cuando el programa se ejecuta en cierta computadora, corresponde al intérprete traducir los comandos de Java en comandos que la computadora local pueda entender.

Las clases de la API de Java permiten a los programadores en Java llevar con más rapidez nuevas aplicaciones al mercado, mediante el uso de componentes preexistentes y comprobados. Esto no sólo reduce el tiempo de desarrollo, sino que también mejora la habilidad del programador en cuanto a depurar y mantener aplicaciones. Para aprovechar las diversas herramientas de Java, es imprescindible que los programadores se familiaricen a sí mismos con la variedad de paquetes y clases en la API de Java. En el sitio `java.sun.com` existen muchos recursos basados en Web, los cuales le pueden ayudar con esta tarea. El principal recurso para aprender acerca de la API de Java es la *documentación de la API de Java*,[11] que puede encontrarse en:

```
java.sun.com/j2se/1.4.1/docs/api/index.html
```

Además, en `java.sun.com` se proporcionan muchos otros recursos, incluyendo tutoriales, artículos y sitios específicos para temas individuales en Java.

Buena práctica de programación 8.7



Evite reinventar la rueda. Estudie las capacidades de la API de Java. Si la API contiene una clase que cumple con los requerimientos de su programa, utilice esa clase en vez de crear una.

Para darnos cuenta de todo el potencial de la reutilización de software, necesitamos mejorar los esquemas de catalogación, esquemas de licenciamiento, los mecanismos de protección que aseguran que no se corrompan las copias maestras de las clases, esquemas de descripción que los diseñadores de sistemas utilizan para determinar si los objetos existentes cumplen con sus necesidades, mecanismos de exploración que determinan cuáles clases están disponibles y qué tan estrechamente cumplen con los requerimientos de los desarrolladores de software, etcétera. Muchos problemas interesantes de investigación y desarrollo se han resuelto, y muchos

11. Puede descargar esta documentación en la página `java.sun.com/j2se/1.4.1/download.html`.

más necesitan resolverse; estos problemas se resolverán debido a que el valor potencial de la reutilización de software es enorme.

8.16 Abstracción de datos y encapsulamiento

Como lo indicamos al principio de este capítulo, las clases normalmente ocultan los detalles de la implementación a los clientes. Esto se conoce como *ocultamiento de información*. Como ejemplo de ello, analicemos una estructura de datos llamada *pila*. Los estudiantes pueden pensar en una pila en términos de una pila de platos. Cuando se coloca un plato sobre la pila, siempre se coloca en la parte superior (a esto se le denomina *empujar* el plato en la pila). De manera similar, cuando se remueve un plato de la pila, siempre se remueve de la parte superior (a esto se le denomina *sacar* el plato de la pila). Las pilas se conocen como *estructuras de datos del tipo último en entrar, primero en salir (UEPS)*; el último elemento que se metió (insertó) en la pila es el primer elemento que se sacó (removió) de ella.

Las pilas pueden implementarse mediante arreglos y con otras estructuras de datos, como las listas enlazadas. (Hablaremos sobre las pilas y las listas enlazadas en el capítulo 20.) El cliente de una clase pila no necesita preocuparse por la implementación de la pila. El cliente sólo sabe que cuando se colocan elementos de datos en la pila, éstos se recuperarán en el orden del último en entrar, primero en salir. El cliente se preocupa acerca de *qué* funcionalidad ofrece una pila, pero no acerca de *cómo* se implementa esa funcionalidad. A este concepto se le conoce como *abstracción de datos*. Aunque los programadores pudieran conocer los detalles de la implementación de una clase, no deben escribir código que dependa de esos detalles. Esto permite que una clase en particular (como una que implemente a una pila y sus operaciones: *meter* y *sacar*) se reemplace con otra versión, sin afectar al resto del sistema. Mientras que los servicios `public` de la clase no cambien (es decir, que cada método original tenga aún el mismo nombre, tipo de valor de retorno y lista de parámetros en la declaración de la nueva clase), el resto del sistema no se ve afectado.

La mayoría de los lenguajes de programación enfatizan las acciones. En estos lenguajes, los datos existen para apoyar las acciones que los programas deben realizar. Los datos son "menos interesantes" que las acciones. Los datos son "crudos". Sólo existen unos cuantos tipos primitivos, y es difícil para los programadores crear sus propios tipos. Java y el estilo orientado a objetos de programación elevan la importancia de los datos. Las principales actividades de la programación orientada a objetos en Java son la creación de tipos (por ejemplo, clases) y la expresión de las interacciones entre objetos de esos tipos. Para crear lenguajes que enfaticen los datos, la comunidad de lenguajes de programación necesitaba formalizar ciertas nociones sobre los datos. La formalización que consideramos aquí es la noción de *tipos de datos abstractos (ADTs)*. Los ADTs reciben, en la actualidad, la misma atención que recibió la programación estructurada en décadas anteriores. Sin embargo, los ADTs no reemplazan a la programación estructurada, sino que proporcionan una formalización adicional para mejorar el proceso de desarrollo de programas.

Considere el tipo primitivo `int`, el cual la mayor parte de las personas lo asociarían con un entero en matemáticas. En vez de ello, un `int` es una representación abstracta de un entero. A diferencia de los enteros matemáticos, los números `int` de computadora tienen un tamaño fijo. Por ejemplo, el tipo `int` en Java está limitado aproximadamente al rango desde –2 mil millones hasta +2 mil millones. Si el resultado de un cálculo queda fuera de este rango se produce un error, y la computadora responde en cierta forma dependiente del equipo. Por ejemplo, podría producir "silenciosamente" un resultado incorrecto como un valor demasiado largo como para caber en una variable `int` (lo que comúnmente se conoce como *desbordamiento aritmético*). Los enteros matemáticos no tienen este problema. Por lo tanto, la noción de un `int` de computadora es solamente una aproximación de la noción de un entero real. Lo mismo se aplica al tipo `float` y a los demás tipos integrados.

Hemos dado por sentado la noción de `int` hasta este punto, pero ahora analicémosla desde una nueva perspectiva. Los tipos como `int`, `float`, `char` y otros son ejemplos de tipos de datos abstractos. Estos tipos son representaciones de nociones reales hasta cierto nivel satisfactorio de precisión, dentro de un sistema computacional.

En realidad, un ADT captura dos nociones: una *representación de datos* y las *operaciones* que pueden realizarse sobre esos datos. Por ejemplo, en Java un `int` contiene un valor entero (datos) y proporciona las operaciones de suma, resta, multiplicación, división y residuo; sin embargo, la división entre cero está prohibida. Los programadores en Java utilizan clases para implementar tipos de datos abstractos.

http://librosysolucionarios.net

Observación de ingeniería de software 8.19



Los programadores pueden crear tipos mediante el uso del mecanismo de clases. Estos nuevos tipos pueden diseñarse de manera que sean tan convenientes de usar como los tipos integrados. Esto marca a Java como un lenguaje extensible. Aunque el lenguaje es fácil de extender mediante nuevos tipos, el programador no puede alterar el lenguaje básico en sí.

Otro de los tipos de datos abstractos que veremos es una *cola* (o "línea de espera"). Los sistemas computacionales utilizan muchas colas internamente. Una cola ofrece un comportamiento bien definido a sus clientes: éstos colocan elementos en una cola, uno a la vez, mediante una operación conocida como *enfilar*, y luego recuperan esos elementos, uno a la vez, mediante una operación conocida como *retirar*. Una cola devuelve los elementos en el orden *primero en entrar, primero en salir (PEPS)*, lo cual significa que el primer elemento insertado en una cola es el primer elemento que se remueve. Conceptualmente, una cola puede volverse infinitamente larga, pero las colas reales son finitas.

La cola oculta una representación de datos interna que lleva el registro de los elementos que esperan actualmente en la línea, y ofrece un conjunto de operaciones a sus clientes (*enfilar* y *retirar*). A los clientes no les preocupa la implementación de la cola; simplemente dependen de que ésta opere "como se indicó". Cuando un cliente enfila a un elemento, la cola debe aceptarlo y colocarlo en algún tipo de estructura de datos PEPS interna. De manera similar, cuando el cliente desea el siguiente elemento de la parte frontal de la cola, ésta debe remover el elemento de su representación interna y entregarlo en orden PEPS (es decir, el elemento que haya estado más tiempo en la cola debe ser el siguiente que se devuelva mediante la siguiente operación de retiro).

El ADT tipo cola garantiza la integridad de su estructura de datos interna. Los clientes no pueden manipular esta estructura de datos directamente; sólo el ADT tipo cola tiene acceso a sus datos internos. Los clientes pueden realizar solamente las operaciones permitidas en la representación de datos; el ADT rechaza las operaciones que su interfaz pública no proporciona.

8.17 (Ejemplo práctico opcional) Acerca de los objetos: Cómo empezar a programar las clases para la simulación del elevador

En las secciones "Acerca de los objetos" de los capítulos 1 a 7, presentamos los fundamentos de la orientación a objetos y desarrollamos un diseño orientado a objetos para nuestra simulación del elevador. En la primera parte de este capítulo presentamos los detalles de la programación con clases en Java. Ahora empezaremos a implementar nuestro diseño orientado a objetos en Java. Al final de esta sección le mostraremos cómo generar código en Java, trabajando a partir de los diagramas de clases. Este proceso se conoce como *ingeniería hacia delante*.[12]

Visibilidad

Ahora aplicaremos *modificadores de acceso* (vea la sección 8.2) a los miembros de nuestras clases. En la sección 8.2 presentamos los modificadores de acceso `public` y `private`. Estos determinan las *visibilidades* o accesibilidades de los atributos y métodos de un objeto, con respecto a otros objetos. Antes de crear los archivos de clase analizaremos qué atributos y métodos de nuestras clases deben ser `public` y cuáles `private`.

En la sección 8.2 hablamos sobre cómo los atributos generalmente debían ser `private` y que los métodos, que son operaciones invocadas por los clientes de una clase dada, generalmente debían de ser `public`. UML emplea *marcadores de visibilidad* para modelar la visibilidad de los atributos y las operaciones. La visibilidad pública se indica colocando un signo positivo (+) antes de una operación o atributo específico; un signo negativo (–) indica una visibilidad privada. En la figura 8.19 se muestra nuestro diagrama de clases actualizado, incluyendo las notaciones de visibilidad.

Navegabilidad

Antes de empezar a implementar nuestro diseño en Java, presentaremos una pieza adicional de notación de UML. El diagrama de clases de la figura 8.20 refina aún más las relaciones entre las clases de la simulación del

12. Booch, G., *The Unified Modeling Language User Guide*, Reading, MA: Addison Wesley Longman, Inc., 1999: 16. (Una vez que existe el código, el proceso de retroceder desde el código para reproducir el diseño se conoce como *ingeniería inversa*.)

elevador, agregando flechas de *navegabilidad* a las líneas de asociación. Las flechas de navegabilidad indican en qué dirección puede navegarse una asociación. Por ejemplo, podemos navegar de la clase `Elevador` a la clase `Timbre`, lo cual permite al `Elevador` sonar el `Timbre`. Sin embargo, y como la flecha de navegabilidad apunta de la clase `Elevador` a la clase `Timbre`, el `Timbre` no puede acceder al `Elevador`. Observe que las asociaciones en este diagrama que no tienen flechas de navegabilidad indican una navegabilidad bidireccional: la navegación puede proceder en cualquier dirección a lo largo de la asociación.[13] Al implementar un sistema diseñado utilizando UML, la navegabilidad ayuda a los programadores a determinar qué objetos necesitan referencias a otros objetos.

Implementación: ingeniería hacia adelante

La ingeniería hacia adelante es el proceso de transformar un diseño, como el que se encuentra en un modelo de UML, para convertirlo en código de un lenguaje de programación específico, como Java. Ahora que hemos hablado sobre la programación de clases en Java aplicaremos ingeniería hacia adelante a las clases en el diagrama de las figuras 8.19 y 8.20, para crear el código para nuestro simulador. El código generado representará el "esqueleto", o la estructura, del simulador.[14] En los capítulos 10 y 11 modificaremos el código para incorporar herencia e interfaces, respectivamente. En los apéndices D, E y F presentaremos el código completo y funcional de Java para nuestro modelo.

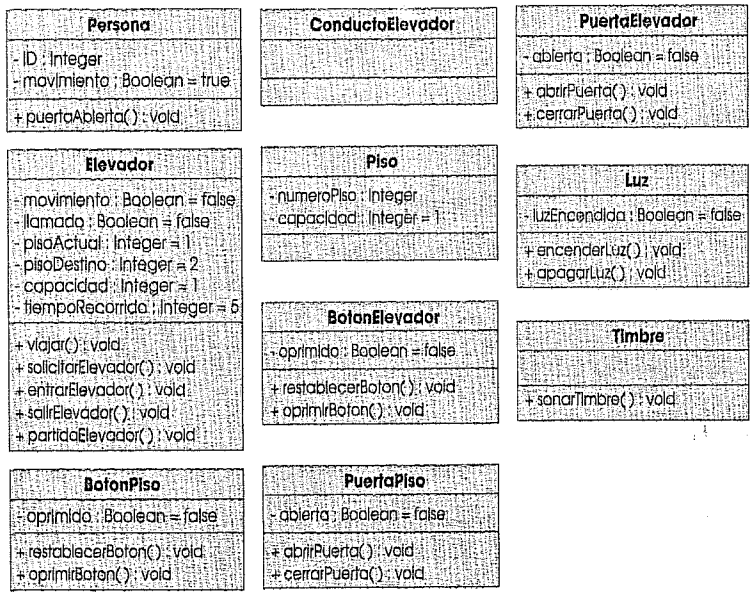


Figura 8.19 Diagrama de clases con notaciones de visibilidad.

13. La especificación de UML versión 1.4 establece que la ausencia de flechas de navegabilidad en una asociación indica, ya sea una navegabilidad bidireccional o una navegabilidad desconocida. En capítulos anteriores no habíamos introducido todavía el concepto de la navegabilidad, por lo que omitimos las flechas para indicar una navegabilidad desconocida. Ahora que hemos introducido la navegabilidad, omitimos las flechas para indicar una navegabilidad bidireccional
14. Hasta ahora hemos presentado sólo aproximadamente la mitad del material del ejemplo práctico; no hemos hablado todavía sobre la herencia, el manejo de eventos, subprocesamiento múltiple o animación. El proceso de desarrollo estándar recomienda terminar el proceso de diseño antes de empezar el proceso de codificación. Técnicamente no habremos terminado el diseño de nuestro sistema sino hasta que hayamos hablado sobre estos temas adicionales, por lo que nuestra implementación actual del código podría parecer prematura. Presentamos solamente una implementación parcial para ilustrar los temas abiertos en el capítulo 8.

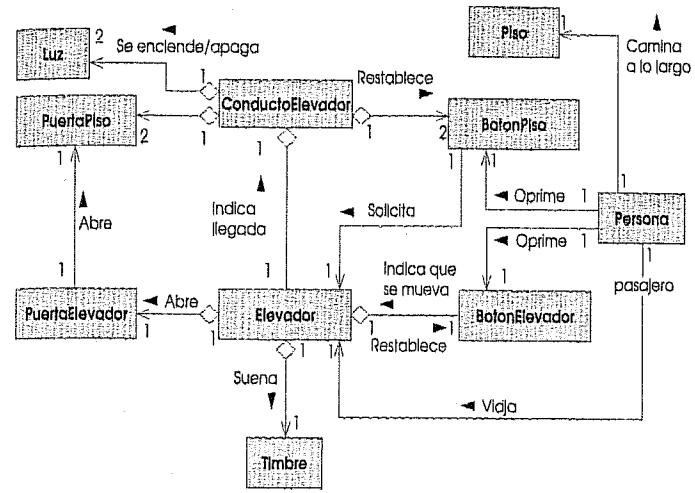


Figura 8.20 Diagrama de clases con navegabilidad.

Como ejemplo aplicaremos ingeniería hacia delante a nuestro diseño de la clase `Elevador`, de la figura 8.19. Emplearemos esta figura para determinar los atributos y operaciones de esa clase. Utilizamos el modelo de UML como se mostró en la figura 3.19, para determinar las asociaciones entre las clases. Vamos a adherirnos a los siguientes cuatro lineamientos para cada clase:

1. Use el nombre localizado en el primer compartimiento para declarar la clase como `public` con un constructor vacío. Por ejemplo, la clase `Elevador` produce

```
public class Elevador {

   public Elevador() {}
}
```

2. Use los atributos localizados en el segundo compartimiento para declarar las variables de instancia. Por ejemplo, los atributos `private` `movimiento`, `llamado`, `pisoActual`, `pisoDestino`, `capacidad` y `tiempoRecorrido` de la clase `Elevador` producen

```
public class Elevador {

   // atributos
   private boolean movimiento;
   private boolean llamado;
   private int pisoActual = 1;
   private int pisoDestino = 2;
   private int capacidad = 1;
   private int tiempoRecorrido = 5;

   // constructor
   public Elevador() {}
}
```

3. Use las asociaciones descritas en el diagrama de clases para declarar las referencias a otros objetos. Por ejemplo, de acuerdo con la figura 3.19, `Elevador` agrega un objeto de cada una de las clases `PuertaElevador`, `BotonElevador` y `Timbre`. Esto produce

http://librosysolucionarios.net

```
public class Elevador {

   // atributos
   private boolean movimiento;
   private boolean llamado;
   private int pisoActual = 1;
   private int pisoDestino = 2;
   private int capacidad = 1;
   private int tiempoRecorrido = 5;

   // objetos asociados
   private PuertaElevador puertaElevador;
   private BotonElevador botonElevador;
   private Timbre timbre;

   // constructor
   public Elevador() {}

}
```

4. Use las operaciones localizadas en el tercer compartimiento de la figura 8.19 para declarar a los métodos. Por ejemplo, las operaciones `public viajar`, `solicitarElevador`, `entrarElevador`, `salirElevador` y `partidaElevador` en la clase `Elevador` producen

```
public class Elevador {

   // atributos
   private boolean movimiento;
   private boolean llamado;
   private int pisoActual = 1;
   private int pisoDestino = 2;
   private int capacidad = 1;
   private int tiempoRecorrido = 5;

   // objetos asociados
   private PuertaElevador puertaElevador;
   private BotonElevador botonElevador;
   private Timbre timbre;

   // constructor
   public Elevador() {}

  // operaciones
   public void viajar() {}
   public void solicitarElevador() {}
   public void entrarElevador() {}
   public void salirElevador() {}
   public void partidaElevador() {}

}
```

Con esto concluimos los fundamentos de la ingeniería hacia delante.

RESUMEN

- La POO (programación orientada a objetos) encapsula los datos (atributos) y métodos (comportamientos) en objetos; los datos y métodos de un objeto están íntimamente entrelazados. La PBO (programación basada en objetos) trata acerca de la creación y el uso de objetos.
- Los objetos tienen la propiedad de ocultar información. Los objetos podrían saber cómo comunicarse entre sí a través de interfaces bien definidas, pero generalmente no se les permite saber cómo están implementados los demás objetos.
- Los programadores en Java crean sus propios tipos llamados clases.
- Los campos no `static` de una clase se conocen como variables de instancia. Los campos `static` de una clase se llaman variables de clase.
- Java usa la herencia para crear nuevas clases a partir de declaraciones de clases existentes.

http://librosysolucionarios.net

- Toda clase en Java es una subclase de `Object`. Por lo tanto, toda nueva clase tiene los atributos y comportamientos de la clase `Object`.
- Las palabras clave `public` y `private` son modificadores de acceso.
- Los campos y métodos declarados con el modificador de acceso `public` son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de la clase en la que están declarados.
- Los campos y métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que están declarados.
- Las variables de instancia se declaran generalmente como `private` y los métodos generalmente se declaran como `public`.
- Los métodos (o servicios) `public` de una clase son utilizados por los clientes de la clase, para manipular los datos almacenados en objetos de esa clase.
- Un constructor tiene el mismo nombre que el de su clase, e inicializa las variables de instancia de un objeto de la clase cuando se crea la instancia de ese objeto. Los constructores pueden sobrecargarse. Pueden tomar argumentos, pero no pueden especificar un tipo de valor de retorno.
- Los constructores y otros métodos que cambian los valores de las variables de instancia deben mantener siempre los datos en un estado consistente.
- Al crearse un objeto, `new` asigna la memoria para el objeto y después llama al constructor para la clase, de manera que se inicialicen las variables de instancia del objeto.
- Dentro del alcance de una clase, los miembros de la clase son accesibles para todos los métodos de esa clase, y pueden referenciarse por su nombre. Fuera del alcance de una clase, los miembros de ésta sólo pueden utilizarse mediante un manejador: una referencia para miembros no `static` o el nombre de la clase para miembros `static`.
- Cada objeto tiene acceso a una referencia a sí mismo (conocida como `this`) que puede usarse dentro de los métodos de la clase para hacer referencia a los campos y métodos del objeto en forma explícita.
- Si no se declaran constructores para una clase, el compilador crea un constructor predeterminado.
- Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los campos y métodos del segundo objeto, inclusive cuando sean `private`.
- Las clases a menudo ofrecen métodos `public` para permitir a los clientes de la clase *establecer* u *obtener* las variables de instancia `private`. Los métodos *establecer* también se conocen comúnmente como métodos mutadores. Los métodos *obtener* también se conocen comúnmente como métodos de acceso o métodos de consulta.
- Todos los eventos tienen un origen: el componente de la GUI con el cual interactuó el usuario para indicar al programa que realizara una tarea.
- Con la composición, un objeto tiene referencias a objetos de otras clases como miembros.
- Si un método declara a una variable con el mismo nombre que un campo de la clase, el campo es sombreado (oculto) por la variable en el método. Una variable de instancia sombreada puede utilizarse en el método si se antepone a su nombre la palabra clave `this` y un punto (`.`).
- Siempre que tenga una referencia en un programa (incluso como resultado de la llamada a un método), esta referencia puede ir seguida por un punto y una llamada a uno de los métodos para el tipo de referencia.
- Java realiza una recolección de basura automática de la memoria. Cuando no hay más referencias a un objeto, éste se marca para la recolección de basura.
- Todas las clases en Java tienen un método llamado `finalize`, el cual no tiene parámetros y no devuelve ningún valor. Este método es llamado por el recolector de basura antes de remover un objeto de la memoria.
- Una variable de clase `static` representa información en toda la clase; todos los objetos de la clase comparten la misma pieza de datos. Los miembros `public static` de una clase pueden utilizarse a través de una referencia a cualquier objeto de esa clase, o pueden utilizarse a través del nombre de la clase, utilizando un punto.
- El método `public static gc`, de la clase `System`, sugiere al recolector de basura que haga su mejor esfuerzo por intentar recolectar objetos basura. No se garantiza que el recolector de basura recolecte objetos en un orden específico (o en algún orden siquiera).
- Un método declarado como `static` no puede tener acceso a los miembros no `static` de la clase. A diferencia de los métodos no `static`, un método `static` no tiene referencia `this`, ya que las variables `static` y los métodos `static` existen en forma independiente de cualquier objeto de una clase.
- Los miembros `static` existen aun cuando no existan objetos de esa clase; están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución.
- Use la palabra clave `final` para especificar que una variable es una constante (no modificable) y que cualquier intento por modificar esa variable es un error. Una variable `final` no puede modificarse una vez que se le asigna un valor por primera vez. Dicha variable debe inicializarse en su declaración, o en cada uno de los constructores de la clase.
- Si los archivos `.class` para las clases utilizadas en un programa se encuentran en el mismo directorio que la clase que las utiliza, no se requieren las declaraciones `import`.

- Cada clase e interfaz en la API de Java pertenecen a un paquete específico, el cual contiene un grupo de clases e interfaces relacionadas. Los paquetes son estructuras de directorios utilizados para organizar clases e interfaces. Los paquetes proporcionan un mecanismo para reutilizar software y una convención para nombres de clase únicos.
- Para crear una clase reutilizable se requiere declarar una clase `public`, agregar una declaración `package` al archivo de declaración de la clase y compilarla en la estructura de directorios del paquete apropiado. Después, la clase puede importarse en un programa.
- Al compilar una clase en un paquete, utilice la opción de línea de comandos `-d` para especificar en dónde se van a crear todos los directorios de la declaración `package`, de manera que el compilador pueda colocar la clase en la estructura de directorios apropiada.
- Los nombres de directorios de los paquetes se vuelven parte del nombre de la clase, cuando ésta se compila. Use este nombre completamente calificado en los programas, o utilice una instrucción `import` con esa clase para poder usar su nombre simple (el nombre de la clase por sí solo) en el programa.
- Cuando no se proporciona un modificador de acceso para un método o variable, cuando ésta se declara en una clase, se considera que el método o variable tiene acceso a nivel de paquete.
- Si un programa utiliza varias clases del mismo paquete, estas clases pueden utilizar los métodos y campos de acceso a nivel de paquete de cada una de las demás clases, a través de una referencia a un objeto.

TERMINOLOGÍA

acceso a nivel de paquete
alcance de una clase
atributo
biblioteca de clases
campo
clase
clase contenedora
cliente de una clase
código reutilizable
comportamiento
composición
constructor
constructor predeterminado
constructor sin argumentos
constructores sobrecargados
control de acceso a los miembros
`-d`, opción del compilador
declaración de clase
desarrollo rápido de aplicaciones (RAD)
encapsulamiento
establecer, método
estado consistente para una variable de instancia
expresión de creación de instancia de clase
`extends`
extensibilidad
`finalize`, método
implementación de una clase
inicializar el objeto de una clase
instancia de una clase
instanciar un objeto de una clase
interfaz para una clase
mensaje
método ayudante
método de acceso
método de clase (`static`)
método de instancia
método mutador
método predicado
método utilitario
modificador de acceso
`new`
Object
obtener, método
ocultamiento de información
`package`, declaración
principio del menor privilegio
`private`
programación basada en objetos (PBO)
programación orientada a objetos (POO)
`public`
`public`, interfaz de una clase
punto (`.`) para el acceso a los miembros
reutilización de software
servicios de una clase
`static`, método
`static`, variable
`this`, referencia
tipo de datos abstracto (ADT)
variable de clase (`static`)
variable de instancia

EJERCICIOS DE AUTOEVALUACIÓN

8.1 Complete las siguientes oraciones:

a) Los miembros de una clase especificados como _____ son accesibles sólo para los métodos de esa clase.
b) Un(a) _____ se utiliza para inicializar las variables de instancia de una clase.
c) Un método _____ se utiliza para asignar valores a las variables de instancia `private` de una clase.
d) Los métodos de una clase generalmente se declaran como _____, y las variables de instancia de una clase generalmente se declaran como _____.
e) Un método _____ se utiliza para recuperar valores de los datos `private` de una clase.
f) La palabra clave _____ introduce la declaración de una clase.
g) Los miembros de una clase especificados como _____ son accesibles en cualquier parte en la que el objeto de una clase esté dentro de su alcance.
h) Un(a) _____ crea un objeto de un tipo especificado y devuelve una _____ a ese objeto.

http://librosysolucionarios.net

i) Una variable _____ representa información disponible para toda la clase.
j) La palabra clave _____ especifica que una variable no puede modificarse una vez que se inicializa.
k) Un método declarado como `static` no puede utilizar los miembros _____ de una clase directamente.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

8.1 a) `private`. b) Constructores. c) *Establecer* (o mutador). d) `public`, `private`. e) *Obtener* (o de acceso). f) `class`. g) `public`. h) Expresión de creación de una instancia de clase, referencia. i) `static` (o de clase). j) `final`. k) No `static`.

EJERCICIOS

8.2 Cree una clase llamada `Complejo` para realizar operaciones aritméticas con números complejos. Estos números tienen la forma

parteReal + *parteImaginaria* * *i*

en donde *i* es

$\sqrt{-1}$

Escriba un programa para probar su clase. Use variables de punto flotante para representar los datos `private` de la clase. Proporcione un constructor que permita que un objeto de esta clase se inicialice al declararse. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos `public` que realicen las siguientes operaciones:

a) Sumar dos números `Complejo`: las partes reales se suman entre sí y las partes imaginarias también se suman entre sí.
b) Restar dos números `Complejo`: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
c) Imprimir números `Complejo` en la forma `(a, b)`, en donde `a` es la parte real y `b` es la imaginaria.

8.3 Cree una clase llamada `Racional` para realizar operaciones aritméticas con fracciones. Escriba un programa para probar su clase. Use variables enteras para representar variables de instancia `private` de la clase: el `numerador` y el `denominador`. Proporcione un constructor que permita a un objeto de esta clase inicializarse al ser declarado. El constructor debe almacenar la fracción en forma reducida; la fracción

`2 / 4`

es equivalente a `1 / 2` y debe guardarse en el objeto como `1` en el `numerador` y `2` en el `denominador`. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos `public` que realicen cada una de las siguientes operaciones:

a) Sumar dos números `Racional`: el resultado de la suma debe almacenarse en forma reducida.
b) Restar dos números `Racional`: el resultado de la resta debe almacenarse en forma reducida.
c) Multiplicar dos números `Racional`: el resultado de la multiplicación debe almacenarse en forma reducida.
d) Dividir dos números `Racional`: el resultado de la división debe almacenarse en forma reducida.
e) Imprimir números `Racional` en la forma `a/b`, en donde `a` es el `numerador` y `b` es el `denominador`.
f) Imprimir números `Racional` en formato de punto flotante. (Considere proporcionar capacidades de formato, que permitan al usuario de la clase especificar el número de dígitos de precisión a la derecha del punto decimal.)

8.4 Modifique la clase `Tiempo3` de la figura 8.7 para incluir el método `tic` que incremente el tiempo almacenado en un objeto `Tiempo3` en un segundo. Además, proporcione un método llamado `incrementarMinuto` para incrementar los minutos y un método `incrementarHora` para incrementar las horas. El objeto `Tiempo3` deberá siempre permanecer en un estado consistente. Escriba un programa que pruebe los métodos `tic`, `incrementarMinuto` e `incrementarHora` para asegurarse de que funcionen correctamente. Asegúrese de probar los siguientes casos:

a) Incrementar al siguiente minuto.
b) Incrementar a la siguiente hora.
c) Incrementar al siguiente día (es decir, de 11:59:59 PM a 12:00:00 AM).

8.5 Modifique la clase `Fecha` de la figura 8.9 para llevar a cabo una comprobación de errores en los valores iniciales para las variables de instancia `mes`, `dia` y `anio` (actualmente sólo valida el mes y el día). Además, pro-

porcione un método llamado `siguienteDia` para incrementar el día en uno. El objeto `Fecha` deberá siempre permanecer en un estado consistente. Escriba un programa para probar el método `siguienteDia` en un ciclo que imprima la fecha durante cada iteración del mismo, para mostrar que el método `siguienteDia` funciona correctamente. Pruebe los siguientes casos:

a) Incrementar al siguiente mes.

b) Incrementar al siguiente año.

8.6 Cree la clase `FechaYHora` que combine la clase `Tiempo3` modificada del ejercicio 8.4 y la clase `Fecha` modificada del ejercicio 8.5. Modifique el método `incrementarHora` para que llame al método `siguienteDia` si la hora se incrementa al día siguiente. Modifique los métodos `aStringEstandar` y `aStringUniversal` para mostrar la fecha junto con la hora. Escriba un programa para probar la nueva clase `FechaYHora`. Pruebe específicamente a la hora de incrementar la hora al siguiente día.

8.7 Modifique los métodos *establecer* en la clase `Tiempo3` de la figura 8.7 para devolver valores de error apropiados si se intenta establecer una de las variables de instancia `hora`, `minuto` o `segundo`, de un objeto de la clase `Tiempo`, a un valor inválido. (*Sugerencia:* Use tipos de valor de retorno `boolean` en cada método.) Escriba un programa que pruebe estos nuevos métodos *establecer* y que produzca mensajes de error cuando se proporcionen valores incorrectos.

8.8 Cree una clase llamada `Rectangulo`. La clase debe tener los atributos `longitud` y `anchura`, cada uno de los cuales debe tener un valor predeterminado de 1. Debe tener también métodos para calcular el `perimetro` y el `area` del rectángulo. Debe tener métodos *establecer* y *obtener* tanto para `longitud` como para `anchura`. Los métodos *establecer* deben verificar que la `longitud` y la `altura` sean números de punto flotante mayores que 0.0 y menores que 20.0. Escriba un programa para probar la clase `Rectangulo`.

8.9 Cree una clase `Rectangulo` más sofisticada que la del ejercicio 8.8. Esta clase debe guardar solamente las coordenadas Cartesianas de las cuatro esquinas del rectángulo. El constructor debe llamar a un método *establecer* que acepte cuatro conjuntos de coordenadas y verifique que cada una de éstas se encuentre en el primer cuadrante. Ninguna coordenada *x* o *y* debe ser mayor de 20.0. El método *establecer* debe también verificar que las coordenadas proporcionadas especifiquen un rectángulo. Proporcione métodos para calcular la `longitud`, `anchura`, `perimetro` y `area`. La longitud será la mayor de las dos dimensiones. Incluya un método predicado llamado `esCuadrado`, el cual determine si el rectángulo es un cuadrado. Escriba un programa para probar la clase `Rectangulo`.

8.10 Cree una clase llamada `EnteroEnorme` la cual utilice un arreglo de 40 elementos representados por dígitos, para guardar enteros de hasta 40 dígitos de longitud cada uno. Proporcione los métodos `entrada`, `salida`, `sumar` y `restar`. Para comparar objetos `EnteroEnorme`, proporcione los siguientes métodos: `esIgualA`, `noEsIgualA`, `esMayorQue`, `esMenorQue`, `esMayorOIgualA`, y `esMenorOIgualA`. Cada uno de estos métodos deberá ser un método predicado que devuelva `true` si la relación se aplica entre los dos objetos `EnteroEnorme`, y `false` si no se aplica. Proporcione un método predicado llamado `esCero`. Si desea hacer algo más, proporcione también los métodos `multiplicar`, `dividir` y `residuo`.

8.11 Cree una clase llamada `TresEnRaya` que le permita escribir un programa complejo para jugar al "tres en raya" (o tres en línea). La clase debe contener un arreglo privado bidimensional de enteros, con un tamaño de 3 por 3. El constructor debe inicializar el tablero vacío con ceros. Permita dos jugadores humanos. Siempre que el primer jugador realice un movimiento, coloque un 1 en el cuadro especificado; coloque un 2 siempre que el segundo jugador realice un movimiento. Cada movimiento debe hacerse en un cuadro vacío. Después de cada movimiento, determine si el juego se ha ganado o si hay un empate. Si desea hacer algo más, modifique su programa de manera que la computadora realice los movimientos para uno de los jugadores. Además, permita que el jugador especifique si desea el primer o segundo turno. Si se siente todavía más motivado, desarrolle un programa que reproduzca un juego de Tres en raya tridimensional, en un tablero de 4 por 4 (*Nota:* ¡Éste es un proyecto retador que podría requerir de muchas semanas de esfuerzo!).

8.12 Explique la noción del acceso a nivel de paquete en Java. Explique los aspectos negativos del acceso a nivel de paquete.

8.13 ¿Qué ocurre cuando se especifica un tipo de valor de retorno, incluso `void`, para un constructor?

8.14 Cree la clase `Fecha` con las siguientes características:

a) Muestre la fecha en varios formatos como:

```
MM/DD/AAAA
Junio 14, 1992
DDD AAAA
```

http://librosysolucionarios.net

b) Use constructores sobrecargados para crear objetos `Fecha` inicializados con fechas en los formatos de la parte (a). En el primer caso, el constructor debe recibir tres valores enteros. En el segundo caso, el constructor debe recibir un objeto `String` y dos valores enteros. En el tercer caso, el constructor debe recibir dos valores enteros, en donde el primer valor representa el número del día en el año. [*Sugerencia:* Para convertir la representación en cadena del mes a un valor numérico, compare cadenas utilizando el método `equals`. Por ejemplo, si `s1` y `s2` son cadenas, la llamada al método `s1.equals( s2 )` devuelve `true` si las cadenas son idénticas; en caso contrario, la llamada al método devuelve `false`.]

8.15 Cree una clase llamada `CuentaDeAhorros`. Use una variable `static` llamada `tasaInteresAnual` para almacenar la tasa de interés anual para todos los cuenta habientes. Cada objeto de la clase debe contener una variable de instancia `private` llamada `saldoAhorros`, que indique la cantidad que el ahorrador tiene actualmente en depósito. Proporcione el método `calcularInteresMensual` para calcular el interés mensual, multiplicando el `saldoAhorros` por la `tasaInteresAnual` dividida entre 12; este interés debe sumarse al `saldoAhorros`. Proporcione un método `static` llamado `modificarTasaInteres` para establecer la `tasaInteresAnual` en un nuevo valor. Escriba un programa para probar la clase `CuentaDeAhorros`. Cree dos instancias de objetos `CuentaDeAhorros`, `ahorrador1` y `ahorrador2`, con saldos de $2000.00 y $3000.00, respectivamente. Establezca la `tasaInteresAnual` en 4%, después calcule el interés mensual e imprima los nuevos saldos para ambos ahorradores. Luego establezca la `tasaInteresAnual` en 5%, calcule el interés del siguiente mes e imprima los nuevos saldos para ambos ahorradores.

8.16 Cree la clase `ConjuntoEnteros`. Cada objeto `ConjuntoEnteros` puede almacenar enteros en el rango de 0 a 100. El conjunto se representa mediante un arreglo de valores `boolean`. El elemento del arreglo `a[i]` es `true` si el entero *i* se encuentra en el conjunto. El elemento del arreglo `a[j]` es `false` si el entero *j* no se encuentra dentro del conjunto. El constructor sin argumentos inicializa el arreglo de Java con el "conjunto vacío" (es decir, un conjunto cuya representación de arreglo contiene valores `false`).

Proporcione los siguientes métodos: el método `union` debe crear un tercer conjunto que sea la unión teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del tercer arreglo se establece en `true` si ese elemento es `true` en cualquiera o en ambos de los conjuntos existentes; en caso contrario, el elemento del tercer conjunto se establece en `false`). El método `interseccion` debe crear un tercer conjunto que sea la intersección teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del arreglo del tercer conjunto se establece en `false` si ese elemento es `false` en uno o ambos de los conjuntos existentes; en caso contrario, el elemento del tercer conjunto se establece en `true`). El método `insertarElemento` debe insertar un nuevo entero *k* en un conjunto (estableciendo `a[k]` en `true`). El método `eliminarElemento` debe eliminar el entero *m* (estableciendo `a[m]` en `false`). El método `aStringConjunto` debe devolver una cadena que contenga un conjunto como una lista de números separados por espacios. Incluya sólo aquellos elementos que estén presentes en el conjunto. Use `- - -` para representar un conjunto vacío. El método `esIgualA` debe determinar si dos conjuntos son iguales. Escriba un programa para probar la clase `ConjuntoEnteros`. Cree instancias de varios objetos `ConjuntoEnteros`. Pruebe que todos sus métodos funcionen correctamente.

8.17 Sería perfectamente razonable para la clase `Tiempo3` de la figura 8.7 representar la hora en forma interna como el número de segundos transcurridos desde medianoche, en vez de usar los tres valores enteros `hora`, `minuto` y `segundo`. Los clientes podrían utilizar los mismos métodos `public` y obtener los mismos resultados. Modifique la clase `Tiempo3` de la figura 8.7 para implementar la clase `Tiempo3` como el número de segundos desde medianoche y mostrar que no hay un cambio visible para los clientes de la clase.

8.18 (*Programa de dibujo.*) Cree un applet de dibujo que dibuje líneas, rectángulos y óvalos al azar. Para este fin, cree un conjunto de clases de figuras "inteligentes", en las que los objetos de estas clases sepan cómo dibujarse a sí mismos si se les proporciona un objeto `Graphics` que les indique en dónde dibujarse (es decir, el objeto `Graphics` permite a una figura dibujarse en el fondo del applet). Los nombres de las clases deben ser `miLinea`, `miRect` y `miOvalo`.

Los datos para la clase `miLinea` deben incluir las coordenadas *x1*, *y1*, *x2* e *y2*. El método `dibujarLinea` de la clase `Graphics` debe conectar los dos puntos suministrados con una línea. Los datos para las clases `miRect` y `miOvalo` deben incluir un valor de coordenada *x* de la esquina superior izquierda, un valor de coordenada *y* de la esquina superior izquierda, una *anchura* (no debe ser negativa) y una *altura* (no debe ser negativa). Todos los datos en cada clase deben ser `private`.

Además de los datos, cada clase debe declarar cuando menos los siguientes métodos `public`:

a) Un constructor sin argumentos que establezca las coordenadas en 0.

b) Un constructor con argumentos que establezca las coordenadas en los valores suministrados.

c) Métodos *establecer* para cada pieza individual de datos, los cuales permitan al programador establecer cualquier pieza de datos en una figura de manera independiente (por ejemplo, si usted tiene una variable de instancia llamada `x1`, debe tener un método llamado `establecerX1`).

d) Métodos *obtener* para cada pieza individual de datos, los cuales permitan al programador recuperar cualquier pieza de datos en una figura de manera independiente (por ejemplo, si usted tiene una variable de instancia llamada `x1`, debe tener un método llamado `obtenerX1`).

e) Un método `dibujar` con la primera línea

```
public void dibujar( Graphics g )
```

que se llamará desde el método `paint` del applet para dibujar una figura en la pantalla.

Si desea proporcionar más métodos para aumentar la flexibilidad, lo puede hacer.

Empiece declarando la clase `MiLinea` y un applet para probar sus clases. El applet deberá tener una variable de instancia `MiLinea` llamada `linea`, que pueda hacer referencia a un objeto `MiLinea` (creado en el método `init` del applet con coordenadas aleatorias). El método `paint` del applet deberá dibujar la figura mediante una instrucción como ésta:

```
linea.dibujar( g );
```

en donde `linea` debe ser la referencia tipo `MiLinea` y `g` debe ser el objeto `Graphics` que será utilizado por la figura para dibujarse a sí misma en el applet.

A continuación, modifique la referencia `MiLinea` para que sea un arreglo de referencias `MiLinea`, y codifique varios objetos `MiLinea` directamente en el programa para dibujarlos. El método `paint` del applet deberá recorrer el arreglo de objetos `MiLinea` y dibujar cada uno de ellos.

Una vez que esté funcionando la parte anterior, deberá declarar las clases `MiOvalo` y `MiRect`, y deberá agregar objetos de estas clases en los arreglos `MiOvalo` y `MiRect`. El método `paint` del applet deberá recorrer cada uno de estos arreglos y dibujar cada una de las figuras. Cree cinco figuras de cada tipo.

Una vez que el applet esté funcionando, seleccione `Volver a cargar (Reload)` en el menú `Subprograma (Applet)` del contenedor de applet para volver a cargar el applet. Esto hará que el applet seleccione nuevos números aleatorios para las figuras, y las dibujará de nuevo.[15]

15. En el capítulo 10 modificaremos este ejercicio para aprovechar las similitudes entre las clases, y así evitar tener varias copias del código.

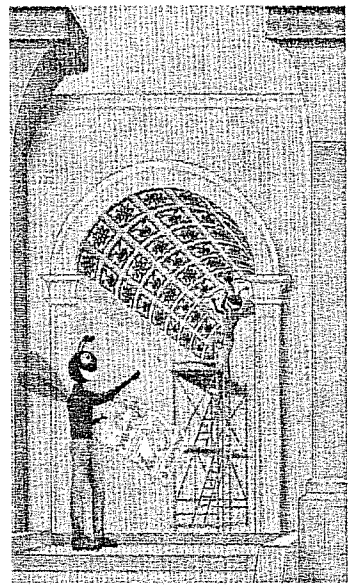
http://librosysolucionarios.net

9

Programación orientada a objetos: Herencia

Objetivos

- Comprender cómo la herencia fomenta la reutilización de software.
- Entender qué son las superclases y las subclases.
- Comprender el uso del modificador de acceso `protected`.
- Utilizar los miembros de superclases mediante `super`.
- Comprender el uso de los constructores y finalizadores en las jerarquías de herencia.
- Presentar un ejemplo práctico que muestre la mecánica de la herencia.



No digas que conoces a alguien por completo, hasta que tengas que dividir una herencia con él.
Johann Kasper Lavater

Este método es para definirse como el número de la clase de todas las clases similares a la clase dada.
Bertrand Russell

Es bueno heredar una biblioteca, pero es mejor coleccionar una.
Augustine Birrell

Plan general

9.1 Introducción
9.2 Superclases y subclases
9.3 Miembros `protected`
9.4 Relación entre las superclases y las subclases
9.5 Ejemplo práctico: Jerarquía de herencia de tres niveles
9.6 Constructores y finalizadores en las subclases
9.7 Ingeniería de software mediante la herencia

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

9.1 Introducción

En este capítulo, comenzamos nuestra discusión acerca de la programación orientada a objetos (POO), introduciendo una de sus características principales, la *herencia*, que es una forma de reutilización de software en la que las clases se crean absorbiendo los datos (atributos) y métodos (comportamientos) de una clase existente, y se mejoran con nuevas capacidades, o con modificaciones en las capacidades ya existentes. La reutilización de software ahorra tiempo durante el desarrollo de programas. También fomenta la reutilización de software probado y depurado de alta calidad, el cual aumenta la probabilidad de que un sistema se implemente con efectividad.

Al crear una clase, en vez de declarar miembros (variables y métodos) completamente nuevos, el programador puede designar que la nueva clase *herede* los miembros de una clase existente. Esta clase existente se conoce como *superclase*, y la nueva clase se conoce como *subclase*. (Otros lenguajes de programación, como C++, se refieren a la superclase como la *clase base* y a la subclase como *clase derivada*.) Una vez creada, cada subclase puede convertirse en superclase de futuras subclases. Una subclase generalmente agrega sus propias variables y métodos. Por lo tanto, una subclase es más específica que su superclase y representa a un grupo más especializado de objetos. Generalmente, la subclase exhibe los comportamientos de su superclase junto con comportamientos adicionales específicos de esta subclase. La *superclase directa* es la superclase a partir de la cual la subclase hereda en forma explícita. Una *superclase indirecta* se hereda de dos o más niveles arriba en la *jerarquía de clases*, la cual define las relaciones de herencia entre las clases. En Java, la jerarquía de clases empieza con la clase `Object` (en el paquete `java.lang`), a partir de la cual heredan todas las clases en Java, ya sea en forma directa o indirecta. En el caso de la *herencia simple*, una clase se deriva de una superclase. Java, a diferencia de C++, no soporta la *herencia múltiple* (que ocurre cuando una clase se deriva de más de una superclase directa).[1]

La experiencia en la creación de sistemas de software nos indica que algunas cantidades considerables de código tratan con casos especiales, estrechamente relacionados. Cuando los programadores se preocupan con casos especiales, los detalles pueden oscurecer el panorama general. Con la programación orientada a objetos, los programadores se enfocan en los elementos comunes entre los objetos en el sistema, en vez de enfocarse en los casos especiales. A este proceso se le conoce como *abstracción*.

Es necesario hacer una diferencia entre la *relación "es un"* y la *relación "tiene un"*. La relación "es un" representa a la herencia; en este tipo de relación, un objeto de una subclase puede tratarse también como un objeto de sus superclases. Por ejemplo, un auto *es un* vehículo. En contraste, la relación "tiene un" identifica a la composición (vea el capítulo 8); en este tipo de relación, un objeto contiene una o más referencias a objetos como miembros. Por ejemplo, un auto *tiene un* volante de dirección.

Las clases nuevas pueden heredar de las clases en diversas *bibliotecas de clases*. Las organizaciones desarrollan sus propias bibliotecas de clases y pueden aprovechar las que ya están disponibles en todo el mundo.

1. En el capítulo 10 explicaremos cómo Java puede usar interfaces para obtener muchos de los beneficios de la herencia múltiple, evitando al mismo tiempo los problemas asociados.

http://librosysolucionarios.net

Es probable que algún día, la mayoría de software nuevo se construya a partir de *componentes reutilizables estándar*, como sucede actualmente con la mayoría del hardware. Esto facilitará el desarrollo de software más poderoso.

9.2 Superclases y subclases

A menudo, un objeto de una clase "es un" objeto de otra clase también. Por ejemplo, en la geometría un rectángulo *es un* cuadrilátero (al igual que los cuadrados, los paralelogramos y los trapezoides). Por lo tanto, en Java puede decirse que la clase `Rectangulo` *hereda* de la clase `Cuadrilatero`. En este contexto, la clase `Cuadrilatero` es una superclase, y la clase `Rectangulo` es una subclase. Un rectángulo *es un* tipo específico de cuadrilátero, pero es incorrecto decir que todo cuadrilátero *es un* rectángulo; el cuadrilátero podría ser un paralelogramo o alguna otra figura. En la figura 9.1 se muestran varios ejemplos sencillos de superclases y subclases.

Como todo objeto de una subclase "es un" objeto de su superclase, y como una superclase puede tener muchas subclases, el conjunto de objetos representados por una superclase generalmente es más grande que el conjunto de objetos representado por cualquiera de sus subclases. Por ejemplo, la superclase `Vehiculo` representa a todos los vehículos, incluyendo autos, camiones, barcos, bicicletas, etcétera. En contraste, la clase `Auto` representa a un subconjunto más pequeño y específico de todos los vehículos.

Las relaciones de herencia forman estructuras jerárquicas en forma de árbol. Una superclase existe en una relación jerárquica con sus subclases. Aunque las clases pueden existir de manera independiente, cuando participan en relaciones de herencia se afilian con otras clases. Una clase se convierte ya sea en una superclase, proporcionando datos y comportamientos a otras clases, o en una subclase, heredando sus datos y comportamientos de otras clases.

Desarrollaremos una jerarquía de clases simple (lo que también se conoce como una *jerarquía de herencia*). Una comunidad universitaria tiene miles de miembros, compuestos por empleados, estudiantes y exalumnos. Los empleados pueden ser miembros del cuerpo docente o administrativo. Los miembros del cuerpo docente pueden ser administradores (como decanos o jefes de departamento) o maestros. Esta estructura organizacional produce la jerarquía de herencia descrita en la figura 9.2. Observe que esta jerarquía de herencia podría contener muchas otras clases. Por ejemplo, los estudiantes pueden ser graduados o no graduados; los no graduados pueden ser de primero, segundo, tercero o cuarto año. Cada flecha en la jerarquía representa una relación "es un". Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir "un `Empleado` *es un* `MiembroDeLaComunidad`" y "un `Maestro` *es un* miembro `Docente`". `MiembroDeLaComunidad` es la *superclase directa* de `Empleado`, `Estudiante` y `Exalumno`, y es una *superclase indirecta* de todas las demás clases en el diagrama. Si comienza desde la parte inferior del diagrama, el lector podrá seguir las flechas y aplicar la relación *es-un* hasta la superclase superior. Por ejemplo, un `Administrador` *es un* miembro `Docente`, *es un* `Empleado` y *es un* `MiembroDeLaComunidad`.

Otra jerarquía de herencia es la jerarquía `Figura` de la figura 9.3. Para especificar que la clase `FiguraBidimensional` se deriva de (o hereda de) la clase `Figura`, la clase `FiguraBidimensional` podría declararse en Java de la siguiente manera:

```
public class FiguraBidimensional extends Figura
```

Superclase	Subclases
`Estudiante`	`EstudianteGraduado, EstudianteNoGraduado`
`Figura`	`Circulo, Triangulo, Rectangulo`
`Prestamo`	`PrestamoAutomovil, PrestamoMejoraCasa, PrestamoHipotecario`
`Empleado`	`Docente, Administrativo`
`CuentaBancaria`	`CuentaDeCheques, CuentaDeAhorros`

Figura 9.1 Ejemplos de herencia.

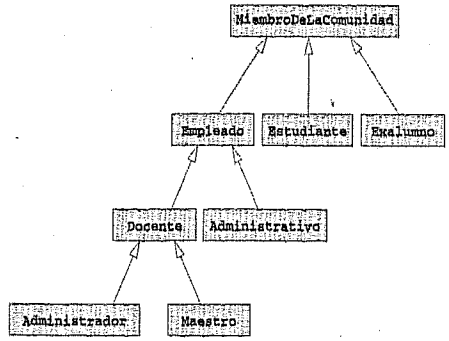


Figura 9.2 Jerarquía de herencia para objetos `MiembroDeLaComunidad` universitaria.

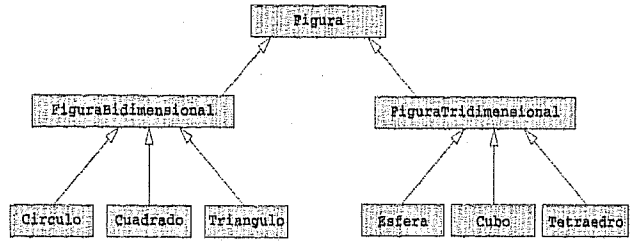


Figura 9.3 Jerarquía de herencia para `Figuras`.

La herencia no es apropiada para todas las relaciones de clases. En el capítulo 8 hablamos sobre la relación *tiene un*, en la que las clases tienen miembros que hacen referencia a los objetos de otras clases.

Tales relaciones crean clases mediante la composición de clases existentes. Por ejemplo, dadas las clases `Empleado`, `FechaDeNacimiento` y `NumeroTelefonico`, no es apropiado decir que un `Empleado` *es una* `FechaDeNacimiento` o que un `Empleado` *es un* `NumeroTelefonico`. Sin embargo, un `Empleado` *tiene una* `FechaDeNacimiento` y también *tiene un* `NumeroTelefonico`.

Es posible tratar a los objetos de superclases y a los objetos de subclases de manera similar; sus similitudes se expresan en los miembros de la superclase. Los objetos de todas las clases que extienden a una superclase común pueden tratarse como objetos de esa superclase (es decir, dichos objetos tienen una relación "es un" con la superclase). Sin embargo, los objetos de una superclase no pueden tratarse como objetos de sus subclases. Por ejemplo, todos los autos son vehículos pero no todos los vehículos son autos. En secciones posteriores consideraremos muchos ejemplos que aprovechan esta relación.

Un problema con la herencia es que una subclase puede heredar métodos que no necesita, o que no debe tener. A pesar de que un método de superclase sea apropiado para una subclase, a menudo esa subclase requiere que el método realice su tarea de una manera específica para la subclase. En dichos casos, la subclase puede *sobrescribir* (redefinir) el método de la superclase con una implementación apropiada.

9.3 Miembros `protected`

En el capítulo 8 hablamos sobre los modificadores de acceso `public` y `private`. Los miembros `public` de una clase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa clase, o una de sus subclases. Los miembros `private` de una clase son accesibles sólo en los métodos de la clase.

Estos miembros `private` de la superclase no se heredan. En esta sección presentaremos el modificador de acceso `protected`. El uso del acceso `protected` ofrece un nivel intermedio de acceso entre `public` y `private`. Los miembros `protected` de una superclase pueden ser utilizados por los miembros de esa superclase, por los miembros de cualquier clase que se derive de esa superclase y por los miembros de otras clases en el mismo paquete (los miembros `protected` también tienen acceso a nivel de paquete).

Todos los miembros `public` y `protected` de una superclase retienen su modificador de acceso original cuando se convierten en miembros de la subclase (por ejemplo, los miembros `public` de la superclase se convierten en miembros `public` de la subclase y, como veremos pronto, los miembros `protected` de la superclase se convierten en miembros `protected` de la subclase). Una subclase puede efectuar cambios de estado en los miembros `private` de la superclase, pero sólo a través de los métodos no `private` que se proporcionen en la superclase y sean heredados por esa subclase.

Observación de ingeniería de software 9.1



Los métodos de una subclase no pueden tener acceso a los miembros `private` de su superclase.

Observación de ingeniería de software 9.2



Declarar datos como `private` ayuda a los programadores a probar, depurar y modificar correctamente los sistemas. Si una subclase pude acceder a los datos `private` de su superclase, las clases que hereden de esa subclase podrían acceder a los datos también. Esto propagaría el acceso a los que debían ser datos privados, y se perderían los beneficios del ocultamiento de la información.

Los métodos de una subclase pueden referirse a los miembros `public` y `protected` que se hereden de la superclase con sólo utilizar los nombres de los miembros. Cuando un método de la subclase sobrescribe al método de la superclase, éste último puede utilizarse desde la subclase si se antepone a su nombre la palabra clave `super` y un punto (`.`). En la sección 9.4 hablaremos sobre el acceso a los miembros sobrescritos de la superclase.

9.4 Relación entre las superclases y las subclases

En esta sección utilizaremos una jerarquía de herencia Punto/Círculo[2] para hablar sobre la relación entre una superclase y una subclase. Dividiremos nuestra discusión de la relación Punto/Círculo en varias partes. Primero declararemos la clase `Punto`, la cual hereda directamente de la clase `Object` y declara como datos `private` a un par de coordenadas *x*-*y*. Después declararemos la clase `Circulo`, la cual hereda directamente de la clase `Object` y declara como datos `private` a un par de coordenadas *x*-*y* (que representan la ubicación del centro del círculo) y un radio. No extenderemos la clase `Punto` para crear la clase `Circulo`; crearemos la clase escribiendo cada una de las líneas de código que se requieran. Después declararemos una clase `Circulo2` separada, que extienda a la clase `Punto` (es decir, un `Circulo2` *es un* `Punto` que también tiene un radio) y trate de acceder a los miembros `private` de la clase `Punto`; esto producirá errores de compilación, ya que la subclase no tiene acceso a los datos `private` de la superclase. Luego mostraremos que, si los datos de `Punto` se declaran como `protected`, una clase `Circulo3` que extienda a la clase `Punto2` podrá acceder a esos datos. Para este fin declararemos la clase `Punto2` con datos `protected`. Ambas clases `Circulo` contienen la misma funcionalidad, pero mostraremos cómo la clase `Circulo3` es más fácil de crear y manejar. Después de hablar sobre la conveniencia de utilizar datos `protected`, regresaremos los datos de `Punto` nuevamente a `private` en la clase `Punto3` (para reforzar la buena ingeniería de software). Luego mostraremos cómo una clase `Circulo4` separada, que extiende a la clase `Punto3`, puede utilizar los métodos de `Punto3` para manipular los datos `private` de `Punto3`.

Creación y uso de una clase `Punto`

Primero examinaremos la declaración de la clase `Punto` (figura 9.4). Recuerde que en la sección 8.2 vimos que toda clase en Java (excepto `Object`) extiende a (o se deriva de) una clase existente. La clase `Punto` no

2. La relación punto/círculo podría parecer antinatural cuando decimos que un círculo "es un" punto, Este ejemplo enseña lo que también se conoce como *herencia estructural*, y se enfoca en la "mecánica" de la herencia y en cómo una superclase y una subclase se relacionan entre sí. En los ejercicios y secciones posteriores, presentaremos más ejemplos de herencia natural.

http://librosysolucionarios.net

especifica a qué clase extiende, por lo que implícitamente extiende a la clase `Object` (paquete `java.lang`). Los servicios `public` de la clase `Punto` incluyen dos constructores (líneas 9 a 12 y líneas 15 a 20) y los métodos `establecerX` (líneas 23 a 26), `obtenerX` (líneas 29 a 32), `establecerY` (líneas 35 a 38), `obtenerY` (líneas 41 a 44) y `toString` (líneas 47 a 50). La clase `Punto` especifica a las variables de instancia `x` y `y` como `private` (líneas 5 y 6), por lo que los objetos de otras clases no pueden acceder a `x` y `y` de manera directa. Técnicamente, aun si las variables de instancia `x` y `y` de `Punto` fueran públicas, `Punto` nunca podría mantener un estado inválido; las variables de instancia `x` y `y` de un objeto `Punto` no pueden contener valores inválidos, ya que el plano de coordenadas *x-y* es infinito en ambas direcciones. No obstante y en general, al declarar las variables de instancia como `private` y al proporcionar métodos *obtener* y *establecer* para manipular y validar las variables de instancia, se refuerza la buena ingeniería de software.

Observación de ingeniería de software 9.3



El compilador de Java establece la superclase de una clase en `Object` cuando el programa no especifica una superclase en forma explícita.

```
1  // Fig. 9.4: Punto.java
2  // La declaración de la clase Punto representa un par de coordenadas x-y.
3
4  public class Punto {
5     private int x; // parte x de un par de coordenadas
6     private int y; // parte y de un par de coordenadas
7
8     // constructor sin argumentos
9     public Punto()
10    {
11       // la llamada implícita al constructor de Object ocurre aquí
12    }
13
14    // constructor
15    public Punto( int valorX, int valorY )
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18       x = valorX;  // no hay necesidad de validación
19       y = valorY;  // no hay necesidad de validación
20    }
21
22    // establecer x en el par de coordenadas
23    public void establecerX( int valorX )
24    {
25       x = valorX;  // no hay necesidad de validación
26    }
27
28    // devuelve x del par de coordenadas
29    public int obtenerX()
30    {
31       return x;
32    }
33
34    // establecer y en el par de coordenadas
35    public void establecerY( int valorY )
36    {
37       y = valorY;  // no hay necesidad de validación
38    }
39
```

Figura 9.4 La clase `Punto` representa un par de coordenadas x-y. (Parte 1 de 2.)

http://librosysolucionarios.net


```
40    // devuelve y del par de coordenadas
41    public int obtenerY()
42    {
43       return y;
44    }
45
46    // devuelve la representación String del objeto Punto
47    public String toString()
48    {
49       return "[" + x + ", " + y + "]";
50    }
51
52 } // fin de la clase Punto
```

Figura 9.4 La clase `Punto` representa un par de coordenadas x-y. (Parte 2 de 2.)

Los constructores no se heredan; por lo tanto, la clase `Punto` no hereda el constructor de la clase `Object`. Sin embargo, los constructores de `Punto` llaman al constructor de la clase `Object` en forma implícita. De hecho, la primera tarea del constructor de cualquier subclase es llamar al constructor de su superclase directa, ya sea en forma implícita o explícita. (Más adelante en esta sección, describiremos la sintaxis para llamar explícitamente al constructor de una superclase.) Si el código no incluye una llamada explícita al constructor de la superclase, Java llama implícitamente al constructor predeterminado o sin argumentos de la superclase. Los comentarios en las líneas 11 y 17 indican en dónde ocurren las llamadas implícitas al constructor predeterminado de la clase `Object`.

El método `toString()` (líneas 47 a 50) es especial. Toda clase en Java (como la clase `Punto`) hereda, ya sea en forma directa o indirecta, de la clase `Object`, la cual es la raíz de la jerarquía de clases en Java. Como lo dijimos anteriormente, esto significa que cada clase hereda los métodos de la clase `Object`. Uno de esos métodos es `toString`, que devuelve una representación `String` de un objeto. En ocasiones el programa llama a este método en forma implícita, como cuando un objeto se concatena a una cadena. El método `toString` original de la clase `Object` es una versión genérica, que se utiliza principalmente para guardar un espacio y puede ser sobrescrito por una subclase (como lo hacemos en los applets con los métodos `init`, `start`, y `paint` de la clase `JApplet`). El método `toString` de la clase `Punto` sobrescribe al método `toString` de la clase `Object`; al ser invocado, el método `toString` de la clase `Punto` devuelve un objeto `String` que contiene un par ordenado de los valores `x` y `y` (línea 49). Para sobrescribir un método de una superclase, una subclase debe declarar un método con la misma firma (nombre y parámetros) que el método en la superclase.

Error común de programación 9.1



Es un error de sintaxis sobrescribir un método con un modificador de acceso más restringido. Un método `public` de la superclase no puede convertirse en un método `protected` o `private` en la subclase.

En la figura 9.5 se prueba la clase `Punto`. En la línea 9 se crea una instancia de un objeto `Punto` llamado `punto` y se pasan al constructor el número `72` como valor para la coordenada *x*, y el número 115 como valor para la coordenada y. En las líneas 12 y 13 se utilizan los métodos `obtenerX` y `obtenerY` de `punto` para recuperar estos valores, y se anexan a la cadena `salida`. En las líneas 15 y 16 se invoca a los métodos `establecerX` y `establecerY` de `punto` para cambiar los valores de las variables de instancia `x` y `y` de `punto`. En la línea 19 se concatena `punto` a una cadena, la cual llama implícitamente al método `toString` de `punto` para obtener la representación de cadena del objeto. Al concatenar una cadena con cualquier objeto se produce una llamada implícita al método `toString` del objeto, para obtener una representación de cadena del mismo; luego se concatenan las cadenas. En la línea 21 se muestra la cadena `salida` al llamar al método `showMessageDialog` de `JOptionPane`.

Creación de una clase `Circulo` sin utilizar herencia

Ahora veremos la segunda parte de nuestra introducción a la herencia, en donde declararemos y probaremos la (completamente nueva) clase `Circulo` (figura 9.6), que contiene un par de coordenadas *x-y* (las cuales indi-

can el centro del círculo) y un radio. Los servicios públicos de la clase **Circulo** incluyen los dos constructores de **Circulo** (líneas 10 a 13 y líneas 16 a 22) y los métodos **establecerX** (líneas 25 a 28), **obtenerX** (líneas 31 a 34), **establecerY** (líneas 37 a 40), **obtenerY** (líneas 43 a 46), **establecerRadio** (líneas 49 a 52), **obtenerRadio** (líneas 55 a 58), **obtenerDiametro** (líneas 61 a 64), **obtenerCircunferencia** (líneas 67 a 70), **obtenerArea** (líneas 73 a 76) y **toString** (líneas 79 a 82). En las líneas 5 a 7 se declaran las variables **x**, **y** y **radio** como variables de instancia **private**. Estas variables y métodos encapsulan todas las características necesarias (es decir, la "geometría analítica") de un círculo. En la sección 9.5 mostraremos cómo este encapsulamiento nos permite reutilizar y extender esta clase.

```
1  // Fig. 9.5: PruebaPunto.java
2  // Prueba de la clase Punto.
3  import javax.swing.JOptionPane;
4
5  public class PruebaPunto {
6
7     public static void main( String[] args )
8     {
9        Punto punto = new Punto( 72, 115 );  // crea objeto Punto
10
11       // obtener las coordenadas del punto
12       String salida = "La coordenada X es " + punto.obtenerX() +
13          "\nLa coordenada Y es " + punto.obtenerY();
14
15       punto.establecerX( 10 );  // establecer la coordenada x
16       punto.establecerY( 20 );  // establecer la coordenada y
17
18       // obtener la representación String del nuevo valor de punto
19       salida += "\n\nLa nueva ubicación de punto es " + punto;
20
21       JOptionPane.showMessageDialog( null, salida ); // mostrar resultados
22
23       System.exit( 0 );
24
25    } // fin de main
26
27 } // fin de la clase PruebaPunto
```

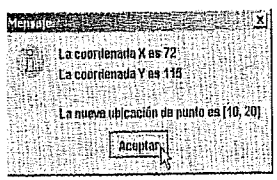


Figura 9.5 Programa de prueba para la clase **Punto**.

```
1  // Fig. 9.6: Circulo.java
2  // La clase Circulo contiene un par de coordenadas x-y y un radio.
3
4  public class Circulo {
5     private int x;          // coordenada x del centro del círculo
6     private int y;          // coordenada y del centro del círculo
7     private double radio;   // radio del círculo
```

Figura 9.6 La clase **Circulo** contiene una coordenada *x-y* y un radio. (Parte 1 de 3.)


```
8
9     // constructor sin argumentos
10    public Circulo()
11    {
12       // la llamada implícita al constructor de Object ocurre aquí
13    }
14
15    // constructor
16    public Circulo( int valorX, int valorY, double valorRadio )
17    {
18       // la llamada implícita al constructor de Object ocurre aquí
19       x = valorX;  // no hay necesidad de validación
20       y = valorY;  // no hay necesidad de validación
21       establecerRadio( valorRadio );
22    }
23
24    // establecer x en el par de coordenadas
25    public void establecerX( int valorX )
26    {
27       x = valorX;  // no hay necesidad de validación
28    }
29
30    // devuelve la x del par de coordenadas
31    public int obtenerX()
32    {
33       return x;
34    }
35
36    // establecer y en el par de coordenadas
37    public void establecerY( int valorY )
38    {
39       y = valorY;  // no hay necesidad de validación
40    }
41
42    // devuelve la y del par de coordenadas
43    public int obtenerY()
44    {
45       return y;
46    }
47
48    // establecer el radio
49    public void establecerRadio( double valorRadio )
50    {
51       radio = ( valorRadio < 0.0 ? 0.0 : valorRadio );
52    }
53
54    // devuelve el radio
55    public double obtenerRadio()
56    {
57       return radio;
58    }
59
60    // calcula y devuelve el diámetro
61    public double obtenerDiametro()
```

Figura 9.6 La clase **Circulo** contiene una coordenada *x-y* y un radio. (Parte 2 de 3.)

http://librosysolucionarios.net

```
62     {
63        return 2 * radio;
64     }
65
66     // calcula y devuelve la circunferencia
67     public double obtenerCircunferencia()
68     {
69        return Math.PI * obtenerDiametro();
70     }
71
72     // calcula y devuelve el área
73     public double obtenerArea()
74     {
75        return Math.PI * radio * radio;
76     }
77
78     // devuelve representación String del objeto Circulo
79     public String toString()
80     {
81        return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
82     }
83
84  } // fin de la clase Circulo
```

Figura 9.6 La clase **Circulo** contiene una coordenada *x-y* y un radio. (Parte 3 de 3.)

```
1  // Fig. 9.7: PruebaCirculo.java
2  // Prueba de la clase Circulo.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaCirculo {
7
8     public static void main( String[] args )
9     {
10       Circulo circulo = new Circulo( 37, 43, 2.5 ); // crea el objeto Circulo
11
12       // obtener las coordenadas x-y y radio iniciales de Circulo
13       String salida = "La coordenada X es " + circulo.obtenerX() +
14          "\nLa coordenada Y es " + circulo.obtenerY() +
15          "\nEl radio es " + circulo.obtenerRadio();
16
17       circulo.establecerX( 35 );          // establecer la nueva coordenada x
18       circulo.establecerY( 20 );          // establecer la nueva coordenada y
19       circulo.establecerRadio( 4.25 );    // establecer el nuevo radio
20
21       // obtener la representación String del nuevo valor de circulo
22       salida += "\n\nLa nueva ubicación y el radio del círculo son\n" +
23          circulo.toString();
24
25       // da formato a los valores de punto flotante con 2 dígitos de precisión
26       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
27
28       // obtener el diámetro del Circulo
```

Figura 9.7 Programa de prueba para la clase **Circulo**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
29       salida += "\nEl diámetro es " +
30          dosDigitos.format( circulo.obtenerDiametro() );
31
32       // obtener la circunferencia del Circulo
33       salida += "\nLa circunferencia es " +
34          dosDigitos.format( circulo.obtenerCircunferencia() );
35
36       // obtener el área del Circulo
37       salida += "\nEl área es " + dosDigitos.format( circulo.obtenerArea() );
38
39       JOptionPane.showMessageDialog( null, salida ); // muestra la salida
40
41       System.exit( 0 );
42
43    } // fin de main
44
45 } // fin de la clase PruebaCirculo
```

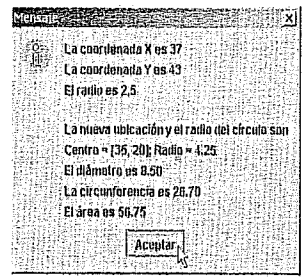


Figura 9.7 Programa de prueba para la clase **Circulo**. (Parte 2 de 2.)

En la figura 9.7 se prueba la clase **Circulo**. En la línea 10 se crea una instancia de un **Circulo**, se pasa al constructor el número **37** como el valor de la coordenada *x*, el número **43** como el valor de la coordenada *y*, y **2.5** como el valor del radio; después se asigna la referencia del objeto a **circulo**. En las líneas 13 a 15 se utilizan los métodos **obtenerX**, **obtenerY** y **obtenerRadio** para recuperar los valores de **circulo**, y se anexan estos valores a la cadena **salida**. En las líneas 17 a 19 se invoca a los métodos **establecerX**, **establecerY** y **establecerRadio** de **circulo** para cambiar las coordenadas *x-y* y el radio, respectivamente. El método **establecerRadio** (figura 9.6, líneas 49 a 52) se asegura de que a la variable de instancia **radio** no se le pueda asignar un valor negativo (es decir, un círculo no puede tener un radio negativo). En las líneas 22 y 23 de la figura 9.7 se hace una llamada al método **toString** de **circulo** en forma explícita, para obtener la representación de cadena del nuevo objeto **circulo**. En las líneas 29 a 37 se hacen llamadas a los métodos **obtenerDiametro**, **obtenerCircunferencia** y **obtenerArea** de **circulo** para obtener el diámetro, la circunferencia y el área de **circulo**, respectivamente. En la línea 39 se muestra la cadena de salida.

Observe que gran parte del código para la clase **Circulo** (figura 9.6) es similar, si no es que idéntico, al código para la clase **Punto** (figura 9.4). Por ejemplo, en la clase **Circulo** las variables de instancia **private x** e **y** y los métodos **establecerX**, **obtenerX**, **establecerY** y **obtenerY** son idénticos a los de la clase **Punto**. Además, los constructores de **Circulo** son casi idénticos a los de la clase **Punto**, con la excepción de que también manipulan el **radio**. Las demás adiciones a la clase **Circulo** son la variable de instancia **private** llamada **radio** y los métodos **establecerRadio**, **obtenerRadio**, **establecerDiametro**, **establecerCircunferencia** y **obtenerArea**.

Parece que copiamos el código de la clase **Punto**, que lo pegamos en la clase **Circulo** y que después modificamos la clase **Circulo** para incluir el radio y los métodos que lo manipulan. Este método de "copiar

y pegar" es a menudo propenso a errores y muy tardado. Peor aún, puede provocar muchas copias físicas del mismo código existente a través de un sistema, creando una pesadilla para mantener el código. ¿Existe alguna manera de "absorber" los atributos y comportamientos de una clase, de manera que los haga parte de otras clases sin tener que duplicar código? En los siguientes ejemplos contestaremos esa pregunta, utilizando un método de construcción de clases más elegante que enfatiza los beneficios de la herencia.

Jerarquía `Punto/Circulo` utilizando herencia

Ahora declararemos y probaremos la clase **`Circulo2`** (figura 9.8), la cual hereda las variables de instancia **`x`** y **`y`**, junto con los métodos **`establecerX`**, **`obtenerX`**, **`establecerY`** y **`obtenerY`** de la clase **`Punto`** (figura 9.4). Un objeto de la clase **`Circulo2`** "es un" **`Punto`** (ya que la herencia le pasa las capacidades de la clase **`Punto`**) pero, según la evidencia que proporciona la clase **`Circulo2`**, también contiene la variable de instancia **`radio`** (figura 9.8, línea 5). La palabra clave **`extends`** en la línea 4 de la declaración de clase indica la herencia. Como una subclase, **`Circulo2`** hereda las variables de instancia y los métodos **`public`** y **`protected`** de la clase **`Punto`**. Los constructores de la clase **`Punto`** no se heredan. Por lo tanto, los servicios públicos de **`Circulo2`** incluyen a los dos constructores de **`Circulo2`** (líneas 8 a 20) (cada clase proporciona sus propios constructores que son específicos para la clase); los métodos **`public`** heredados de la clase **`Punto`**; los métodos **`establecerRadio`** y **`obtenerRadio`** (líneas 23 a 32); y los métodos **`obtenerDiametro`**, **`obtenerCircunferencia`**, **`obtenerArea`** y **`toString`** (líneas 35 a 57).

```
1  // Fig. 9.8: Circulo2.java
2  // La clase Circulo2 hereda de Punto.
3
4  public class Circulo2 extends Punto {
5     private double radio;  // El radio de Circulo2
6
7     // constructor sin argumentos
8     public Circulo2()
9     {
10       // la llamada implícita al constructor de Punto ocurre aquí
11    }
12
13    // constructor
14    public Circulo2( int valorX, int valorY, double valorRadio )
15    {
16       // la llamada implícita al constructor de Punto ocurre aquí
17       x = valorX;  // no permitido: x es privada en Punto
18       y = valorY;  // no permitido: y es privada en Punto
19       establecerRadio( valorRadio );
20    }
21
22    // establecer el radio
23    public void establecerRadio( double valorRadio )
24    {
25       radio = ( valorRadio < 0.0 ? 0.0 : valorRadio );
26    }
27
28    // return radio
29    public double obtenerRadio()
30    {
31       return radio;
32    }
33
34    // calcular y devolver el diámetro
```

Figura 9.8 Los miembros **`private`** de la superclase no son accesibles en la subclase. (Parte 1 de 2.)


```
35    public double obtenerDiametro()
36    {
37       return 2 * radio;
38    }
39
40    // calcular y devolver la circunferencia
41    public double obtenerCircunferencia()
42    {
43       return Math.PI * obtenerDiametro();
44    }
45
46    // calcular y devuolver el área
47    public double obtenerArea()
48    {
49       return Math.PI * radio * radio;
50    }
51
52    // devolver la representación String de un objeto Circulo
53    public String toString()
54    {
55       // uso de x y y no permitido: x y y son privadas en Punto
56       return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
57    }
58
59 } // fin de la clase Circulo2
```

```
Circulo2.java:17: x has private access in Punto
      x = valorX;  // no permitido: x es privada en Punto
      ^
Circulo2.java:18: y has private access in Punto
      y = valorY;  // no permitido: y es privada en Punto
      ^
Circulo2.java:56: x has private access in Punto
      return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
                            ^
Circulo2.java:56: y has private access in Punto
      return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
                                   ^
4 errors
```

Figura 9.8 Los miembros **`private`** de la superclase no son accesibles en la subclase. (Parte 2 de 2.)

El constructor sin argumentos (líneas 8 a 11) llama implícitamente al constructor sin argumentos de la clase **`Punto`**; después Java establece implícitamente las coordenadas *x-y* en 0. El otro constructor (líneas 14 a 20) llama implícitamente al constructor sin argumentos de la clase **`Punto`** y después establece la coordenada *x-y* en un valor específico. El compilador genera errores de sintaxis para las líneas 17 y 18 (y la línea 56, en donde el método **`toString`** de **`Circulo2`** trata de utilizar los valores de **`x`** y **`y`** directamente), ya que la subclase **`Circulo2`** no ha heredado, y por lo tanto, no se le permite acceder a las variables de instancia **`private`** **`x`** e **`y`** de la superclase **`Punto`**.

Jerarquía `Punto/Circulo` utilizando datos `protected`

Para permitir a la clase **`Circulo2`** acceder a las variables de instancia **`x`** y **`y`** de la superclase, podemos declarar esos miembros como **`protected`** en la superclase. Como dijimos en la sección 9.3, los miembros

http://librosysolucionarios.net

protected de una superclase son heredados por cualquier subclase de esa superclase. La clase **Punto2** (figura 9.9) es una modificación de la clase **Punto** (figura 9.4), la cual declara las variables de instancia **x** y **y** como **protected** (figura 9.9, líneas 5 y 6) en lugar de **private**. Además de haber cambiado el nombre de la clase (y por consecuencia, haber cambiado el nombre del constructor) a **Punto2**, el resto de la declaración de clase de la figura 9.9 es idéntico a la figura 9.4.

```
1  // Fig. 9.9: Punto2.java
2  // La declaración de la clase Punto2 representa un par de coordenadas x-y.
3
4  public class Punto2 {
5     protected int x; // parte x de un par de coordenadas
6     protected int y; // parte y de un par de coordenadas
7
8     // constructor sin argumentos
9     public Punto2()
10    {
11       // la llamada implícita al constructor de Object ocurre aquí
12    }
13
14    // constructor
15    public Punto2( int valorX, int valorY )
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18       x = valorX;  // no hay necesidad de validación
19       y = valorY;  // no hay necesidad de validación
20    }
21
22    // establecer x en el par de coordenadas
23    public void establecerX( int valorX )
24    {
25       x = valorX;  // no hay necesidad de validación
26    }
27
28    // devolver x del par de coordenadas
29    public int obtenerX()
30    {
31       return x;
32    }
33
34    // establecer y en el par de coordenadas
35    public void establecerY( int valorY )
36    {
37       y = valorY;  // no hay necesidad de validación
38    }
39
40    // devolver y del par de coordenadas
41    public int obtenerY()
42    {
43       return y;
44    }
45
46    // devolver la representación String del objeto Punto2
47    public String toString()
48    {
```

Figura 9.9 La clase **Punto2** representa un par de coordenadas *x-y* como datos **protected**. (Parte 1 de 2.)


```
49       return "[" + x + ", " + y + "]";
50    }
51
52 } // fin de la clase Punto2
```

Figura 9.9 La clase **Punto2** representa un par de coordenadas *x-y* como datos **protected**. (Parte 2 de 2.)

La clase **Circulo3** (figura 9.10) es una modificación de la clase **Circulo2** (figura 9.8) que extiende a la clase **Punto2** en vez de **Punto**. Como la clase **Circulo3** extiende a la clase **Punto2** (línea 5), los objetos de la clase **Circulo3** heredan las variables de instancia que se declararon como **protected** en la clase **Punto2** (es decir, las variables **x** y **y**). Como resultado, el compilador no genera errores al compilar el constructor de **Circulo3** en las líneas 15 a 21, y el método **toString** en las líneas 54 a 57, ya que **x** y **y** ahora son miembros de la clase **Circulo3**. Los objetos de una subclase también heredan los miembros **protected** de cualquiera de las superclases indirectas de esa subclase.

```
1  // Fig. 9.10: Circulo3.java
2  // La clase Circulo3 hereda de Punto2 y tiene acceso a los miembros
3  // protected x y y de Punto2.
4
5  public class Circulo3 extends Punto2 {
6     private double radio;  // el radio de Circulo3
7
8     // constructor sin argumentos
9     public Circulo3()
10    {
11       // la llamada implícita al constructor de Punto2 ocurre aquí
12    }
13
14    // constructor
15    public Circulo3( int valorX, int valorY, double valorRadio )
16    {
17       // la llamada implícita al constructor de Punto2 ocurre aquí
18       x = valorX;  // no hay necesidad de validación
19       y = valorY;  // no hay necesidad de validación
20       establecerRadio( valorRadio );
21    }
22
23    // establecer el radio
24    public void establecerRadio( double valorRadio )
25    {
26       radio = ( valorRadio < 0.0 ? 0.0 : valorRadio );
27    }
28
29    // return radio
30    public double obtenerRadio()
31    {
32       return radio;
33    }
34
35    // calcular y devolver el diámetro
36    public double obtenerDiametro()
37    {
```

Figura 9.10 Clase **Circulo3** que hereda los datos **protected** de **Punto2**. (Parte 1 de 2.)

http://librosysolucionarios.net

```
38          return 2 * radio;
39       }
40
41       // calcular y devolver la circunferencia
42       public double obtenerCircunferencia()
43       {
44          return Math.PI * obtenerDiametro();
45       }
46
47       // calcular y devolver el área
48       public double obtenerArea()
49       {
50          return Math.PI * radio * radio;
51       }
52
53       // devolver la representación String de un objeto Circulo3
54       public String toString()
55       {
56          return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
57       }
58
59    } // fin de la clase Circulo3
```

Figura 9.10 Clase **Circulo3** que hereda los datos **protected** de **Punto2**. (Parte 2 de 2.)

La clase **Circulo3** no hereda los constructores de la clase **Punto2**. Sin embargo, el constructor sin argumentos de la clase **Circulo3** (líneas 9 a 12) llama al constructor sin argumentos de la clase **Punto2** en forma implícita. Recuerde que la primera tarea del constructor de cualquier subclase es llamar al constructor de su superclase directa, ya sea en forma implícita o explícita. Si el código no incluye una llamada explícita al constructor de la superclase, se hace una llamada implícita al constructor sin argumentos de la superclase. El constructor de las líneas 9 a 12 llama primero al constructor sin argumentos de **Punto2**, y después Java inicializa las variables heredadas y establece **radio** en sus valores predeterminados de cero.

En la figura 9.11 se realizan en la clase **Circulo3** pruebas idénticas a las que se realizaron en la figura 9.7 para la clase **Circulo** (figura 9.6). Observe que los resultados de los dos programas son idénticos. Aunque declaramos a la clase **Circulo** sin utilizar la herencia y declaramos a la clase **Circulo3** utilizando la herencia, ambas clases proporcionan la misma funcionalidad. El código fuente para la clase **Circulo3**, que consta de 59 líneas, es considerablemente más corto que el código fuente de la clase **Circulo**, que es de 84 líneas, ya que la clase **Circulo3** hereda parte de su funcionalidad de **Punto2**, mientras que la clase **Circulo** no hereda ninguna funcionalidad. Además, ahora hay solamente una copia de la funcionalidad del punto que se mencionó en la clase **Punto2**. Esto hace que el código sea más fácil de depurar, mantener y modificar, ya que el código relacionado con el punto existe sólo en el archivo de la figura 9.9.

```
1  // Fig. 9.11: PruebaCirculo3.java
2  // Prueba de la clase Circulo3.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaCirculo3 {
7
8     public static void main( String[] args )
9     {
```

Figura 9.11 Los miembros **protected** de la superclase se heredan en la subclase **Circulo3**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
10       // crear instancia de objeto Circulo
11       Circulo3 circulo = new Circulo3( 37, 43, 2.5 );
12
13       // obtener coordenadas x-y y radio iniciales de Circulo3
14       String salida = "La coordenada X es " + circulo.obtenerX() +
15          "\nLa coordenada Y es " + circulo.obtenerY() +
16          "\nEl radio es " + circulo.obtenerRadio();
17
18       circulo.establecerX( 35 );        // establecer la nueva coordenada x
19       circulo.establecerY( 20 );        // establecer la nueva coordenada y
20       circulo.establecerRadio( 4.25 );  // establecer el nuevo radio
21
22       // obtener representación String del nuevo valor de circulo
23       salida += "\n\nLa nueva ubicación y radio de circulo son\n" +
24          circulo.toString();
25
26       // dar formato a los valores de punto flotante con 2 dígitos de precisión
27       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
28
29       // obtener el diámetro del Circulo
30       salida += "\nEl diámetro es " +
31          dosDigitos.format( circulo.obtenerDiametro() );
32
33       // obtener la circunferencia del Circulo
34       salida += "\nLa circunferencia es " +
35          dosDigitos.format( circulo.obtenerCircunferencia() );
36
37       // obtener el área del Circulo
38       salida += "\nEl área es " + dosDigitos.format( circulo.obtenerArea() );
39
40       JOptionPane.showMessageDialog( null, salida ); // mostrar salida
41
42       System.exit( 0 );
43
44    } // fin del método main
45
46 } // fin de la clase PruebaCirculo3
```

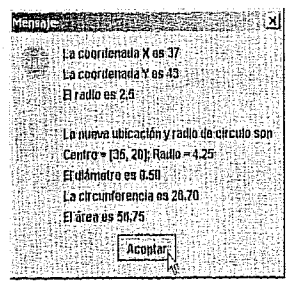


Figura 9.11 Los miembros **protected** de la superclase se heredan en la subclase **Circulo3**. (Parte 2 de 2.)

En este ejemplo declaramos las variables de instancia de la superclase como **protected**, para que las subclases pudieran heredarlas. La herencia de las variables de instancia protegidas permite un ligero aumento en el rendimiento, ya que evitamos incurrir en la sobrecarga de una llamada a un método *establecer* u *obtener*.

Sin embargo, dichos aumentos en el rendimiento son a menudo insignificantes, en comparación con las optimizaciones que los compiladores pueden realizar. Es mejor utilizar datos `private` para fomentar la buena ingeniería de software. Su código será más fácil de mantener, modificar y depurar.

El uso de variables de instancia `protected` crea dos problemas. En primer lugar, el objeto de la subclase no tiene que usar un método para establecer el valor de la variable. Por lo tanto, el objeto de una subclase puede asignar un valor inválido a la variable, dejando así al objeto en un estado inconsistente. Por ejemplo, si declaráramos la variable de instancia `radio` de `Circulo3` como `protected`, el objeto de una subclase (por ejemplo, `Cilindro`) podría entonces asignar un valor negativo a `radio`. El segundo problema debido al uso de variables de instancia `protected` es que hay una mayor probabilidad de que los métodos de la subclase se escriban de manera que dependan en la implementación de la subclase. En la práctica, las subclases deben depender solamente de los servicios de la superclase (es decir, de los métodos que no sean `private`), y no de su implementación. Con variables de instancia `protected` en la superclase, si cambia la implementación de la superclase, tal vez tengamos que modificar a todas las subclases de esa superclase. Por ejemplo, si por alguna razón cambiáramos los nombres de las variables de instancia `x` y `y` a `coordenadaX` y `coordenadaY`, entonces tendríamos que hacerlo en todas las ocurrencias en las que una subclase hiciera referencia a estas variables de instancia de la superclase directamente. En tal caso, se dice que el software es *frágil* o *quebradizo*, ya que un pequeño cambio en la superclase puede "quebrantar" la implementación de la subclase. El programador debe ser capaz de modificar la implementación de la clase libremente, sin tener que dejar de proporcionar los mismos servicios a las subclases. (Claro que, si cambian los servicios de la superclase debemos reimplementar nuestras subclases, pero el buen diseño orientado a objetos trata de evitar esto.)



Buena práctica de programación 9.1

Use el modificador de acceso `protected` cuando una superclase deba proporcionar un servicio (es decir, un método) sólo a sus subclases y no a otros clientes.



Buena práctica de programación 9.2

Declare a las variables de instancia de la superclase como `private` (en vez de `protected`) para que la implementación de la superclase pueda cambiar sin afectar a las implementaciones de las subclases.



Tip para prevenir errores 9.1

Siempre que sea posible, evite incluir variables de instancia `protected` en una superclase. En vez de ello, incluya métodos no `private` que utilicen a las variables de instancia `private`, para asegurar que el objeto mantenga un estado consistente.

Jerarquía `Punto/Circulo` utilizando datos `private`

Ahora reexaminaremos una vez más nuestro ejemplo de la jerarquía `Punto/Circulo`, tratando esta vez de utilizar las mejores prácticas de ingeniería de software. En la clase `Punto3` (figura 9.12) se declaran las variables de instancia `x` y `y` como `private` (líneas 5 y 6), y se exponen los métodos `establecerX`, `obtenerX`, `establecerY`, `obtenerY` y `toString` para manipular estos valores. La subclase `Circulo4` (figura 9.13) hereda estos métodos no `private` de la superclase (`establecerX`, `obtenerX`, `establecerY` y `obtenerY`) y puede acceder a los miembros `private` de la superclase mediante estos métodos.

```
1  // Fig. 9.12: Punto3.java
2  // La declaración de la clase Punto representa un par de coordenadas x-y.
3
4  public class Punto3 {
5     private int x;  // parte x de un par de coordenadas
6     private int y;  // parte y de un par de coordenadas
7
8     // constructor sin argumentos
9     public Punto3()
```

Figura 9.12 La clase `Punto3` utiliza métodos para manipular sus datos `private`. (Parte 1 de 2.)

http://librosysolucionarios.net


```
10    {
11       // la llamada implícita al constructor de Object ocurre aquí
12    }
13
14    // constructor
15    public Punto3( int valorX, int valorY )
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18       x = valorX;  // no hay necesidad de validación
19       y = valorY;  // no hay necesidad de validación
20    }
21
22    // establecer x en el par de coordenadas
23    public void establecerX( int valorX )
24    {
25       x = valorX;  // no hay necesidad de validación
26    }
27
28    // devolver x del par de coordenadas
29    public int obtenerX()
30    {
31       return x;
32    }
33
34    // establecer y en el par de coordenadas
35    public void establecerY( int valorY )
36    {
37       y = valorY;  // no hay necesidad de validación
38    }
39
40    // devolver y del par de coordenadas
41    public int obtenerY()
42    {
43       return y;
44    }
45
46    // devolver la representación String del objeto Punto3
47    public String toString()
48    {
49       return "[" + obtenerX() + ", " + obtenerY() + "]";
50    }
51
52 } // fin de la clase Punto3
```

Figura 9.12 La clase `Punto3` utiliza métodos para manipular sus datos `private`. (Parte 2 de 2.)

La clase `Circulo4` (figura 9.13) tiene varias modificaciones en las implementaciones de sus métodos, que la diferencian de la clase `Circulo3` (figura 9.10). El constructor de la clase `Circulo4` (líneas 16 a 20) invoca a un constructor de `Punto3` para inicializar la porción correspondiente a la superclase de un objeto `Circulo4` (es decir, las variables `x` y `y` heredadas de `Punto3`). En realidad, Java requiere que el constructor de una subclase llame al constructor de su superclase para inicializar las variables de instancia de la superclase que son heredadas por la subclase. El constructor sin argumentos de las líneas 10 a 13 no llama a un constructor de `Punto3` en forma explícita, por lo que Java llama automáticamente al constructor sin argumentos de la clase `Punto3` (declarado en las líneas 9 a 12 de la figura 9.12).

```
1  // Fig. 9.13: Circulo4.java
2  // La clase Circulo4 hereda de Punto3 y accede a las variables x y y
3  // de Punto 3 mediante los métodos public de Punto3.
4
5  public class Circulo4 extends Punto3 {
6
7     private double radio;  // El radio de Circulo4
8
9     // constructor sin argumentos
10    public Circulo4()
11    {
12       // La llamada implícita al constructor de Punto3 ocurre aquí
13    }
14
15    // constructor
16    public Circulo4( int valorX, int valorY, double valorRadio )
17    {
18       super( valorX, valorY ); // llamar explícitamente al constructor de Punto3
19       establecerRadio( valorRadio );
20    }
21
22    // establecer el radio
23    public void establecerRadio( double valorRadio )
24    {
25       radio = ( valorRadio < 0.0 ? 0.0 : valorRadio );
26    }
27
28    // devolver el radio
29    public double obtenerRadio()
30    {
31       return radio;
32    }
33
34    // calcular y devolver el diámetro
35    public double obtenerDiametro()
36    {
37       return 2 * obtenerRadio();
38    }
39
40    // calcular y devolver la circunferencia
41    public double obtenerCircunferencia()
42    {
43       return Math.PI * obtenerDiametro();
44    }
45
46    // calcular y devolver el área
47    public double obtenerArea()
48    {
49       return Math.PI * obtenerRadio() * obtenerRadio();
50    }
51
52    // devolver la representación String del objeto Circulo4
53    public String toString()
54    {
```

Figura 9.13 La clase **Circulo4** que extiende a **Punto3** y que no proporciona datos **protected**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
55       return "Centro = " + super.toString() + "; Radio = " + obtenerRadio();
56    }
57
58 } // fin de la clase Circulo4
```

Figura 9.13 La clase **Circulo4** que extiende a **Punto3** y que no proporciona datos **protected**. (Parte 2 de 2.)

En la línea 18, el segundo constructor de **Circulo4** invoca al constructor de **Punto3** (declarado en las líneas 15 a 20 de la figura 9.12), utilizando la *sintaxis de llamada al constructor de la superclase*: la palabra clave **super**, seguida por un conjunto de paréntesis que contienen los argumentos del constructor de la superclase. Estos argumentos son **valorX** y **valorY**, que se utilizan para inicializar a los miembros **x** y **y** de la superclase. Si el constructor de **Circulo4** no invoca al constructor de **Punto3** en forma explícita, entonces se invoca el constructor sin argumentos de **Punto3** en forma implícita y Java establece las variables **x** y **y** en sus valores predeterminados de 0. Si la clase **Punto3** no proporciona un constructor sin argumentos, el compilador produce un error. La llamada al constructor de la superclase debe ser la primera instrucción en el cuerpo del constructor de la subclase. Use **super()** para llamar al constructor sin argumentos de la superclase en forma explícita.

Error común de programación 9.2

Se produce un error de compilación si el constructor de una subclase llama a uno de los constructores de su superclase con argumentos que no concuerdan exactamente con el número y tipos de parámetros especificados en una de las declaraciones de constructor de la superclase.

Observe que cada uno de los métodos **obtenerDiametro** (figura 9.13, líneas 35 a 38), **obtenerArea** (líneas 47 a 50) y **toString** (líneas 53 a 56) invocan al método **obtenerRadio** para obtener el valor del radio, en vez de acceder directamente a la variable **radio**. Si decidimos renombrar la variable de instancia **radio**, sólo necesitaremos cambiar los cuerpos de los métodos **establecerRadio** y **obtenerRadio**.

El método **toString** de la clase **Circulo4** (figura 9.13, líneas 53 a 56) sobrescribe el método **toString** de la clase **Punto3** (figura 9.12, líneas 47 a 50). La nueva versión muestra los valores de las variables de instancia **private x** y **y** de la clase **Punto3** mediante una llamada al método **toString** de **Punto3** con la expresión **super.toString()** (figura 9.13, línea 55). Observe la sintaxis utilizada para invocar, desde una subclase, a un método sobrescrito de la superclase: se colocan la palabra clave **super** y un punto (.) antes del nombre del método de la superclase. Esta invocación al método es una buena práctica de ingeniería de software; recuerde que, si un método realiza todas o algunas de las acciones necesarias para otro método, es mejor llamar a ese método en vez de duplicar su código. Al hacer que el método **toString** de **Circulo4** invoque al método **toString** de **Punto3** para realizar parte de la tarea de devolver la representación de cadena de un objeto **Circulo4** (es decir, para mostrar los valores de las coordenadas *x* y *y*), se evita que se duplique código y se reducen los problemas de mantenimiento.

Error común de programación 9.3

*Cuando se sobrescribe un método de la superclase en una subclase, a menudo la versión de la subclase llama a la versión de la superclase para realizar trabajo adicional. Si no se anteponen al nombre del método de la superclase la palabra clave **super** y un punto (.) cuando se haga referencia al método de la superclase, se produce una recursividad infinita, ya que el método de la subclase se llamaría entonces a sí mismo.*

Error común de programación 9.4

*Al declarar un método **public** o **protected** de una superclase con una lista de parámetros distinta en la subclase, se oculta la versión de la superclase de ese método. Los intentos por llamar a la versión de la superclase resultan en errores de compilación.*

En la figura 9.14 se realizan manipulaciones en un objeto **Circulo4** de manera idéntica a las que se hicieron en la figura 9.7, y en la figura 9.11 de las clases **Circulo** y **Circulo3**, respectivamente. Aunque cada clase de "círculo" se comporta de manera idéntica, la clase **Circulo4** cuenta con la mejor ingeniería. Mediante el uso de la herencia, hemos construido de manera efectiva y eficiente, una clase con buena ingeniería.

```
1  // Fig. 9.14: PruebaCirculo4.java
2  // Prueba de la clase Circulo4.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaCirculo4 {
7
8     public static void main( String[] args )
9     {
10       // crear instancia de objeto Circulo
11       Circulo4 circulo = new Circulo4( 37, 43, 2.5 );
12
13       // obtener las coordenadas x-y y radio iniciales de Circulo4
14       String salida = "La coordenada X es " + circulo.obtenerX() +
15          "\nLa coordenada Y es " + circulo.obtenerY() +
16          "\nEl radio es " + circulo.obtenerRadio();
17
18       circulo.establecerX( 35 );          // establecer la nueva coordenada x
19       circulo.establecerY( 20 );          // establecer la nueva coordenada y
20       circulo.establecerRadio( 4.25 );    // establecer el nuevo radio
21
22       // obtener la representación String del nuevo valor de circulo
23       salida += "\n\nLa nueva ubicación y radio de circulo son\n" +
24          circulo.toString();
25
26       // dar formato a los valores de punto decimal con 2 dígitos de precisión
27       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
28
29       // obtener el diámetro del Circulo
30       salida += "\nEl diámetro es " +
31          dosDigitos.format( circulo.obtenerDiametro() );
32
33       // obtener la circunferencia del Circulo
34       salida += "\nLa circunferencia es " +
35          dosDigitos.format( circulo.obtenerCircunferencia() );
36
37       // obtener el área de Circulo
38       salida += "\nEl área es " + dosDigitos.format( circulo.obtenerArea() );
39
40       JOptionPane.showMessageDialog( null, salida ); // mostrar resultados
41
42       System.exit( 0 );
43
44    } // fin de main
45
46 } // fin de la clase PruebaCirculo4
```

Figura 9.14 Los datos **private** de la superclase son accesibles para una subclase, mediante los métodos **public** o **protected** heredados por la subclase. (Parte 1 de 2.)

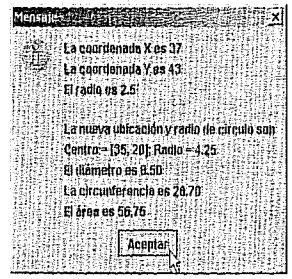


Figura 9.14 Los datos **private** de la superclase son accesibles para una subclase, mediante los métodos **public** o **protected** heredados por la subclase. (Parte 2 de 2.)

9.5 Ejemplo práctico: Jerarquía de herencia de tres niveles

Consideremos ahora un ejemplo de herencia más substancial, en el que se involucra una jerarquía Punto/Circulo de tres niveles. En la sección 9.4 desarrollamos las clases **Punto3** (figura 9.12) y **Circulo4** (figura 9.13). Ahora presentaremos un ejemplo en el que derivaremos la clase **Cilindro** de la clase **Circulo4**.

La primera clase que emplearemos en nuestro ejemplo práctico es **Punto3** (figura 9.12). Declaramos las variables de instancia de **Punto3** como **private**. La clase **Punto3** también contiene los métodos **establecerX**, **obtenerX**, **establecerY** y **obtenerY** para acceder a **x** y **y**, y el método **toString** para devolver la representación de cadena de un objeto **Punto3**.

También utilizaremos la clase **Circulo4** (figura 9.13), la cual extiende a la clase **Punto3**. La clase **Circulo4** contiene la funcionalidad de la clase **Punto3** y proporciona el método **establecerRadio** (para asegurar que la variable de instancia **radio** no pueda guardar un valor negativo) y los métodos **obtenerRadio**, **obtenerDiametro**, **obtenerCircunferencia**, **obtenerArea** y **toString**. Las subclases de **Circulo4** (como **Cilindro**, que presentaremos en unos momentos) deben sobrescribir estos métodos según sea necesario, para proporcionar implementaciones que sean específicas para la subclase. Por ejemplo, un círculo tiene un área que se calcula mediante la fórmula πr^2, en donde r representa el radio del círculo. Sin embargo, un cilindro tiene un área superficial que se calcula mediante la fórmula $(2\pi r^2) + (2\pi rh)$, en donde r representa el radio del cilindro y h representa su altura. Por lo tanto, la clase **Cilindro** debe sobrescribir el método **obtenerArea**, para reflejar este cálculo.

En la figura 9.15 presentamos la clase **Cilindro**, la cual hereda de la clase **Circulo4**. La clase **Cilindro** especifica que un **Cilindro** tiene una **altura** (línea 5). Los servicios públicos de la clase **Cilindro** incluyen a los métodos **establecerRadio**, **obtenerRadio**, **obtenerDiametro**, **obtenerCircunferencia**, **obtenerArea** y **toString** que hereda de la clase **Circulo4**; los métodos **establecerX**, **obtenerX**, **establecerY** y **obtenerY** heredados indirectamente de **Punto3**; los constructores de **Cilindro** (líneas 8 a 19); y los métodos **establecerAltura**, **obtenerAltura**, **obtenerArea**, **obtenerVolumen** y **toString** (líneas 22 a 49) de **Cilindro**. Los métodos **obtenerArea** y **toString** sobrescriben a los métodos con el mismo nombre, que se heredan de la clase **Circulo4**.

```
1  // Fig. 9.15: Cilindro.java
2  // Clase Cilindro que hereda de Circulo4.
3
4  public class Cilindro extends Circulo4 {
5     private double altura;  // altura de Cilindro
6
```

Figura 9.15 La clase **Cilindro** hereda de **Circulo4** y sobrescribe el método **obtenerArea**. (Parte 1 de 2.)

http://librosysolucionarios.net

```
7     // constructor sin argumentos
8     public Cilindro()
9     {
10       // la llamada implícita al constructor de Circulo4 ocurre aquí
11    }
12
13    // constructor
14    public Cilindro( int valorX, int valorY, double valorRadio,
15       double valorAltura )
16    {
17       super( valorX, valorY, valorRadio ); // llamar al constructor de Circulo4
18       establecerAltura( valorAltura );
19    }
20
21    // establecer la altura del Cilindro
22    public void establecerAltura( double valorAltura )
23    {
24       altura = ( valorAltura < 0.0 ? 0.0 : valorAltura );
25    }
26
27    // obtener la altura del Cilindro
28    public double obtenerAltura()
29    {
30       return altura;
31    }
32
33    // sobrescribir al método obtenerArea de Circulo4 para calcular área del Cilindro
34    public double obtenerArea()
35    {
36       return 2 * super.obtenerArea() + obtenerCircunferencia() * obtenerAltura();
37    }
38
39    // calcular el volumen del Cilindro
40    public double obtenerVolumen()
41    {
42       return super.obtenerArea() * obtenerAltura();
43    }
44
45    // devolver la representación String del objeto Cilindro
46    public String toString()
47    {
48       return super.toString() + "; Altura = " + obtenerAltura();
49    }
50
51 } // fin de la clase Cilindro
```

Figura 9.15 La clase **Cilindro** hereda de **Circulo4** y sobrescribe el método **obtenerArea**. (Parte 2 de 2.)

El método **obtenerArea** (líneas 34 a 37) sobrescribe el método **obtenerArea** de **Circulo4** para calcular el área superficial del cilindro. El método **toString** (líneas 46 a 49) sobrescribe el método **toString** de la clase **Circulo4** para devolver la representación **String** del cilindro. La clase **Cilindro** también declara el método **obtenerVolumen** (líneas 40 a 43) para calcular el volumen del cilindro.

En la figura 9.16 se muestra una aplicación llamada **PruebaCilindro** que prueba la clase **Cilindro**. En la línea 11 se crea una instancia de un objeto **Cilindro** llamado **cilindro**. En las líneas 14 a 16 se utilizan los métodos **obtenerX**, **obtenerY**, **obtenerRadio** y **obtenerAltura** de **cilindro** para

obtener información acerca de este objeto, ya que **PruebaCilindro** no puede acceder directamente a las variables de instancia **private** de la clase **Cilindro**. En las líneas 18 a 21 se utilizan los métodos **establecerX**, **establecerY**, **establecerRadio** y **establecerAltura** para restablecer las coordenadas *x*-*y* de **cilindro** (suponemos que las coordenadas *x*-*y* del cilindro especifican la posición del centro de su parte inferior, en el plano *x*-*y*), su radio y su altura. La clase **Cilindro** puede utilizar los métodos **establecerX**, **obtenerX**, **establecerY** y **obtenerY**, ya que la clase **Cilindro** los hereda indirectamente de la clase **Punto3**. (La clase **Cilindro** hereda los métodos **establecerX**, **obtenerX**, **establecerY** y **obtenerY** directamente de la clase **Circulo4**, que a su vez los hereda directamente de la clase **Punto3**.) En las líneas 24 a 26 se invoca el método **toString** de **cilindro** para obtener la representación **String** del objeto **cilindro**. En las líneas 32 y 33, y en las líneas 36 y 37 se invocan los métodos **obtenerDiametro** y **obtenerCircunferencia**, respectivamente, del objeto **Cilindro** (como la clase **Cilindro** hereda los métodos de la clase **Circulo4**, se invocan estos métodos, exactamente como se declararon en **Circulo4**). En las líneas 40 a 43 se invocan los métodos **obtenerArea** y **obtenerVolumen**, respectivamente, para obtener el área superficial y el volumen del **cilindro**. En la línea 45 se muestra la cadena resultante.

```
1  // Fig. 9.16: PruebaCilindro.java
2  // Prueba de la clase Cilindro.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaCilindro {
7
8     public static void main( String[] args )
9     {
10       // crear el objeto Cilindro
11       Cilindro cilindro = new Cilindro( 12, 23, 2.5, 5.7 );
12
13       // obtener las coordenadas x-y, radio y altura iniciales del Cilindro
14       String salida = "La coordenada X es " + cilindro.obtenerX() +
15          "\nLa coordenada Y es " + cilindro.obtenerY() + "\nEl radio es " +
16          cilindro.obtenerRadio() + "\nLa altura es " + cilindro.obtenerAltura();
17
18       cilindro.establecerX( 35 );          // establecer la nueva coordenada x
19       cilindro.establecerY( 20 );          // establecer la nueva coordenada y
20       cilindro.establecerRadio( 4.25 );    // establecer el nuevo radio
21       cilindro.establecerAltura( 10.75 ); // establecer la nueva altura
22
23       // obtener la representación String del nuevo valor de cilindro
24       salida +=
25          "\n\nLa nueva ubicación, radio y altura del cilindro son\n" +
26          cilindro.toString();
27
28       // dar formato a los valores de punto flotante con 2 dígitos de precisión
29       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
30
31       // obtener el diámetro del Cilindro
32       salida += "\n\nEl diámetro es " +
33          dosDigitos.format( cilindro.obtenerDiametro() );
34
35       // obtener la circunferencia del Cilindro
36       salida += "\nLa circunferencia es " +
37          dosDigitos.format( cilindro.obtenerCircunferencia() );
38
```

Figura 9.16 Programa de prueba para la jerarquía **Punto/Circulo/Cilindro**. (Parte 1 de 2.)

http://librosysolucionarios.net

```
39          // obtener el área del Cilindro
40          salida += "\nEl área es " + dosDigitos.format( cilindro.obtenerArea() );
41
42          // obtener el volumen del Cilindro
43          salida += "\nEl volumen es " + dosDigitos.format( cilindro.obtenerVolumen() );
44
45          JOptionPane.showMessageDialog( null, salida ); // mostrar resultados
46
47          System.exit( 0 );
48
49       } // fin de main
50
51    } // fin de la clase PruebaCilindro
```

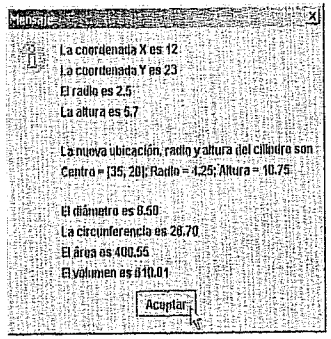


Figura 9.16 Programa de prueba para la jerarquía `Punto/Circulo/Cilindro`. (Parte 2 de 2.)

Utilizando el ejemplo Punto/Circulo/Cilindro, hemos demostrado el uso y los beneficios de la herencia. Pudimos desarrollar las clases `Circulo4` y `Cilindro` con mucha mayor rapidez mediante el uso de la herencia que si las hubiéramos desarrollado "desde cero". La herencia evita la duplicación de código y los problemas de mantenimiento de código asociados.

9.6 Constructores y finalizadores en las subclases

Como explicamos en la sección anterior, al crear la instancia del objeto de una subclase se genera una cadena de llamadas a los constructores, en donde el constructor de la subclase, antes de realizar sus propias tareas, invoca al constructor de su superclase directa, ya sea en forma explícita (mediante la referencia `super`) o implícita (llamando al constructor predeterminado de la superclase, o a su constructor sin argumentos). De manera similar, si la superclase se deriva de otra clase, el constructor de la superclase tiene que invocar al constructor de la siguiente clase que esté un nivel arriba en la jerarquía, y así sucesivamente. El último constructor llamado en la cadena siempre es el constructor de la clase `Object`. El cuerpo del constructor de la subclase original termina de ejecutarse en último lugar. Cada uno de los constructores de la superclase manipula a las variables de instancia de la superclase, las cuales son heredadas por el objeto de la subclase. Por ejemplo, considere de nuevo la jerarquía `Punto3/Circulo4/Cilindro` de las figuras 9.12, 9.13 y 9.15. Cuando un programa crea un objeto `Cilindro`, se realiza una llamada al constructor de `Cilindro`. Ese constructor llama al constructor de `Circulo4`, el cual a su vez llama al constructor de `Punto3`. El constructor de `Punto3` llama al constructor de `Object`. El constructor de la clase `Object` tiene un cuerpo vacío, por lo que devuelve inmediatamente el control al constructor de `Punto3`, el cual manipula entonces las coordenadas *x-y* del objeto `Cilindro`. Cuando el constructor de `Punto3` termina de ejecutarse, devuelve el control al constructor de `Circulo4`, el cual manipula el radio del objeto `Cilindro`. Cuando el constructor de `Circulo4` termina de ejecutarse, devuelve el control al constructor de `Cilindro`, el cual manipula la altura del objeto `Cilindro`.

http://librosysolucionarios.net

Observación de ingeniería de software 9.4



Cuando un programa crea el objeto de una subclase, el constructor de la subclase llama inmediatamente al constructor de la superclase (en forma explícita mediante `super`*, o en forma implícita). El cuerpo del constructor de la superclase se ejecuta para manipular las variables de instancia de la superclase, y el cuerpo del constructor de la subclase se ejecuta para manipular las variables de instancia de la subclase. Java se asegura de que todas las variables de instancia se inicialicen con sus valores predeterminados. Por lo tanto, aunque un constructor no asigne un valor a una variable de instancia, la variable se inicializa de todas formas.*

Si las clases en su jerarquía de clases tienen sus propios métodos `finalize`, el método `finalize` de la subclase debe invocar al método `finalize` de la superclase como su última acción, para asegurar que todas las partes de un objeto se finalicen apropiadamente en caso de que el recolector de basura reclame la memoria de ese objeto. Cuando se llama al método `finalize` de un objeto de la subclase, este método realiza su tarea y luego debe invocar al método `finalize` de la siguiente superclase en la jerarquía. Este proceso debe repetirse hasta que se llame al finalizador de la superclase `Object`. Después la memoria se vuelve disponible para ser utilizada por nuevos objetos.

En nuestro siguiente ejemplo utilizaremos nuevamente la jerarquía Punto/Circulo, declarando una nueva clase `Punto` (figura 9.17) y una nueva clase `Circulo` (figura 9.18). Cada una de estas clases declara constructores y métodos `finalize`, los cuales imprimen mensajes al ser invocados.[3]

La clase `Punto` (figura 9.17) contiene las mismas características que la versión de la clase que se muestra en la figura 9.4. Modificamos los constructores (líneas 9 a 13 y 16 a 23) y agregamos un método `finalize` (líneas 26 a 29). Cada uno de ellos imprime una línea de texto en la línea de comandos, al momento de su invocación. Observe la concatenación de `this` con una cadena en las líneas 12, 22 y 28, en la cual se invoca implícitamente el método `toString` del objeto para obtener la representación de cadena de ese objeto.

```
1  // Fig. 9.17: Punto.java
2  // La declaración de la clase Punto representa un par de coordenadas x-y.
3
4  public class Punto {
5     private int x;  // parte x del par de coordenadas
6     private int y;  // parte y del par de coordenadas
7
8     // constructor sin argumentos
9     public Punto()
10    {
11       // la llamada implícita al constructor de Object ocurre aquí
12       System.out.println( "Constructor sin argumentos de Punto: " + this );
13    }
14
15    // constructor
16    public Punto( int valorX, int valorY )
17    {
18       // la llamada implícita al constructor de Object ocurre aquí
19       x = valorX;  // no hay necesidad de validación
20       y = valorY;  // no hay necesidad de validación
21
22       System.out.println( "Constructor de Punto: " + this );
23    }
24
25    // finalizador
```

Figura 9.17 La superclase `Punto` contiene dos constructores y un finalizador. (Parte 1 de 2.)

[3] Al igual que en la sección 8.11, utilizamos el método `finalize` en este ejemplo sólo para mostrar cuándo el recolector de basura remueve un objeto de la memoria. Generalmente no se recomienda el uso de `finalize`.

```
26    protected void finalize()
27    {
28       System.out.println( "Finalizador de Punto: " + this );
29    }
30
31    // establecer x en el par de coordenadas
32    public void establecerX( int valorX )
33    {
34       x = valorX;  // no hay necesidad de validación
35    }
36
37    // devolver x del par de coordenadas
38    public int obtenerX()
39    {
40       return x;
41    }
42
43    // establecer y en el par de coordenadas
44    public void establecerY( int valorY )
45    {
46       y = valorY;  // no hay necesidad de validación
47    }
48
49    // devolver y del par de coordenadas
50    public int obtenerY()
51    {
52       return y;
53    }
54
55    // devolver la representación String del objeto Punto
56    public String toString()
57    {
58       return "[" + obtenerX() + ", " + obtenerY() + "]";
59    }
60
61 } // fin de la clase Punto
```

Figura 9.17 La superclase **Punto** contiene dos constructores y un finalizador. (Parte 2 de 2.)

La clase **Circulo** (figura 9.18) contiene las características de la clase **Circulo4** (figura 9.13). Modificamos los constructores (líneas 9 a 13 y 16 a 22 de la figura 9.18) y agregamos un método **finalize** (figura 9.18, líneas 25 a 30). Cada uno de ellos imprime una línea de texto en la línea de comandos, al momento de su invocación. De nuevo, observe la concatenación de **this** con una cadena en las líneas 12, 21 y 27 para obtener la representación de cadena del objeto.

```
1  // Fig. 9.18: Circulo.java
2  // Declaración de la clase Circulo.
3
4  public class Circulo extends Punto {
5
6     private double radio;  // el radio del Circulo
7
8     // constructor sin argumentos
```

Figura 9.18 La clase **Circulo** que extiende a **Punto**. (Parte 1 de 3.)

http://librosysolucionarios.net


```
9     public Circulo()
10    {
11       // la llamada implícita al constructor de Punto ocurre aquí
12       System.out.println( "Constructor sin argumentos de Circulo: " + this );
13    }
14
15    // constructor
16    public Circulo( int valorX, int valorY, double valorRadio )
17    {
18       super( valorX, valorY );  // llamar al constructor de Punto
19       establecerRadio( valorRadio );
20
21       System.out.println( "Constructor de Circulo: " + this );
22    }
23
24    // finalizador
25    protected void finalize()
26    {
27       System.out.println( "Finalizador de Circulo: " + this );
28
29       super.finalize();  // llamar al método finalize de la superclase
30    }
31
32    // establecer el radio
33    public void establecerRadio( double valorRadio )
34    {
35       radio = ( valorRadio < 0.0 ? 0.0 : valorRadio );
36    }
37
38    // devolver el radio
39    public double obtenerRadio()
40    {
41       return radio;
42    }
43
44    // calcular y devolver el diámetro
45    public double obtenerDiametro()
46    {
47       return 2 * obtenerRadio();
48    }
49
50    // calcular y devolver la circunferencia
51    public double obtenerCircunferencia()
52    {
53       return Math.PI * obtenerDiametro();
54    }
55
56    // calcular y devolver el área
57    public double obtenerArea()
58    {
59       return Math.PI * obtenerRadio() * obtenerRadio();
60    }
61
62    // devolver la representación String del objeto Circulo
63    public String toString()
```

Figura 9.18 La clase **Circulo** que extiende a **Punto**. (Parte 2 de 3.)

```
64     {
65        return "Centro = " + super.toString() + "; Radio = " + obtenerRadio();
66     }
67
68  } // fin de la clase Circulo
```

Figura 9.18 La clase `Circulo` que extiende a `Punto`. (Parte 3 de 3.)

```
1  // Fig. 9.19: PruebaConstructorFinalizador.java
2  // Muestra el orden en que se hacen las llamadas a los constructores
3  // y finalizadores de la superclase y la subclase.
4
5  public class PruebaConstructorFinalizador {
6
7     public static void main( String args[] )
8     {
9        Punto punto;
10       Circulo circulo1, circulo2;
11
12       punto = new Punto( 11, 22 );
13
14       System.out.println();
15       circulo1 = new Circulo( 72, 29, 4.5 );
16
17       System.out.println();
18       circulo2 = new Circulo( 5, 7, 10.67 );
19
20       punto = null;     // se marca para la recolección de basura
21       circulo1 = null;  // se marca para la recolección de basura
22       circulo2 = null;  // se marca para la recolección de basura
23
24       System.out.println();
25
26       System.gc();   // llamar al recolector de basura
27
28    } // fin de main
29
30 } // fin de la clase PruebaConstructorFinalizador
```

```
Constructor de Punto: [11, 22]

Constructor de Punto: Centro = [72, 29]; Radio = 0.0
Constructor de Circulo: Centro = [72, 29]; Radio = 4.5

Constructor de Punto: Centro = [5, 7]; Radio = 0.0
Constructor de Circulo: Centro = [5, 7]; Radio = 10.67

Finalizador de Punto: [11, 22]
Finalizador de Circulo: Centro = [72, 29]; Radio = 4.5
Finalizador de Punto: Centro = [72, 29]; Radio = 4.5
Finalizador de Circulo: Centro = [5, 7]; Radio = 10.67
Finalizador de Punto: Centro = [5, 7]; Radio = 10.67
```

Figura 9.19 Orden de la llamada a un constructor y a un finalizador.

http://librosysolucionarios.net

En la figura 9.19 se muestra el orden en el que se hacen las llamadas a los constructores y finalizadores para los objetos de clases que forman parte de una jerarquía de herencia. El método `main` (líneas 7 a 28) empieza creando la instancia de un objeto `Punto` (línea 12). Después, en la línea 15 se crea la instancia de un objeto `Circulo` llamado `circulo1`. Esta acción invoca al constructor de `Punto` para que imprima un mensaje con los valores que recibe del constructor de `Circulo`, y después se imprime el mensaje especificado en el constructor de `Circulo`. En la línea 18 se crea la instancia de un objeto `Circulo` llamado `circulo2`. De nuevo, se hace una llamada al constructor de `Punto` y otra al constructor de `Circulo`. Observe que, en cada caso, el cuerpo del constructor de `Punto` se ejecuta antes que el cuerpo del constructor de `Circulo`.

En las líneas 20 a 22 se establecen `punto`, `circulo1` y `circulo2` en `null`. Los objetos a los que estas variables hacen referencia ya no son necesarios en el programa, por lo que Java marca a estos objetos para la recolección de basura. Java garantiza que, antes de que el recolector de basura se ejecute para reclamar el espacio de cada uno de estos objetos, se harán las llamadas a los métodos `finalize` para cada objeto. No obstante, recuerde que en la sección 8.10 vimos que no se garantiza cuándo se ejecutará realmente el recolector de basura. Por esta razón, invocamos al método estático `gc` de la clase `System` en la línea 26. Java no garantiza el orden en el que el recolector de basura reclamará los objetos; por lo tanto, no puede garantizar cuál de los finalizadores se ejecutará primero. Cuando se realiza la llamada al método `System.gc`, se realizan las llamadas a los finalizadores de los objetos `punto`, `circulo1` y `circulo2`. El finalizador de `Circulo` y el finalizador de `Punto` son llamados (en ese orden) para el objeto `circulo1`, y los finalizadores de `Circulo` y `Punto` se llaman (en ese orden) para el objeto `circulo2`. (*Nota:* El método `finalize` de la clase `Object` no hace nada, por lo que no lo llamamos desde el método `finalize` de la clase `Punto`.)

9.7 Ingeniería de software mediante la herencia

En esta sección hablaremos sobre la personalización del software existente mediante la herencia. Cuando una nueva clase extiende a una clase existente, la nueva clase hereda los miembros no `private` de la clase existente. Podemos personalizar la nueva clase para que se adapte a nuestras necesidades, incluyendo miembros adicionales y sobrescribiendo los miembros de la superclase. Esto se hace en Java sin necesidad de que el programador de la subclase tenga que modificar el código fuente de la superclase. Java simplemente requiere acceso al archivo `.class` de la superclase, de manera que pueda compilar y ejecutar cualquier programa que utilice o extienda a la superclase. Esta poderosa capacidad es atractiva para los fabricantes independientes de software (ISVs), que pueden desarrollar clases propietarias para vender o licenciar, y poner a disposición de los usuarios esas clases en formato de código objeto. De esta forma, los usuarios pueden derivar nuevas clases de estas bibliotecas de clases con rapidez y sin necesidad de acceder al código fuente propietario de los ISVs.

En ocasiones es difícil para los estudiantes apreciar el alcance de los problemas a los que se enfrentan los diseñadores que trabajan en proyectos de software de gran escala en la industria. Las personas que tienen experiencia en esos proyectos dicen que la reutilización efectiva de software mejora el proceso de desarrollo del mismo. La programación orientada a objetos facilita la reutilización del software, reduciendo en consecuencia el tiempo de desarrollo.

La disponibilidad de bibliotecas de clases substanciales y útiles produce los máximos beneficios de la reutilización de software a través de la herencia. El interés en las bibliotecas de clases está creciendo en forma exponencial. Al igual que las pequeñas versiones de software producidas por los ISVs se convirtieron en una industria de crecimiento explosivo con la llegada de la computadora personal, así también sucedió con la creación y venta de bibliotecas de clases. Los diseñadores de aplicaciones construyen sus aplicaciones con estas bibliotecas, y los diseñadores de bibliotecas son recompensados al tener sus bibliotecas incluidas con las aplicaciones. Las bibliotecas de clases estándar de Java que se incluyen con la J2SE 1.4.0 tienden a ser de propósito general. Sin embargo, existe un compromiso mundial masivo en cuanto al desarrollo de bibliotecas de clases para una enorme variedad de aplicaciones.



Observación de ingeniería de software 9.5

En la etapa de diseño de un sistema orientado a objetos, el diseñador a menudo encuentra que ciertas clases están íntimamente relacionadas. El diseñador debe "resaltar" los atributos y comportamientos comunes, y colocarlos en una superclase. Entonces, el diseñador debe utilizar la herencia para desarrollar subclases, otorgándoles capacidades más allá de las que se hereden de la superclase.



Observación de ingeniería de software 9.6

El proceso de declarar una subclase no afecta al código fuente de su superclase. La herencia preserva la integridad de una superclase.



Observación de ingeniería de software 9.7

Así como los diseñadores de sistemas no orientados a objetos deben evitar una proliferación de métodos, los diseñadores de sistemas orientados a objetos deben evitar la proliferación de las clases. Esta proliferación crea problemas administrativos y puede dificultar la reutilización de software, ya que al cliente se le dificulta localizar la clase más apropiada en una enorme biblioteca de clases. La alternativa es crear menos clases que proporcionen una funcionalidad más substancial, pero dichas clases podrían llegar a proveer demasiada funcionalidad.



Tip de rendimiento 9.1

Si las subclases son más grandes de lo necesario (es decir, que contengan demasiada funcionalidad), los recursos de memoria y de procesamiento podrían desperdiciarse. Extienda la clase con la funcionalidad que se "aproxime más" a lo que se necesita.

Leer las declaraciones de las subclases puede ser algo confuso, ya que los miembros heredados no se declaran en forma explícita en las subclases, pero son miembros de ellas. Un problema similar existe al tratar de documentar los miembros de la subclase.

En secciones anteriores presentamos la herencia (la habilidad de crear clases absorbiendo los miembros de una clase existente y embelleciéndolos con nuevas capacidades). En el capítulo 10 nos basaremos en nuestra discusión acerca de la herencia para presentar el *polimorfismo* (un concepto orientado a objetos que nos permite escribir programas que manejen, de una manera más general, una amplia variedad de clases relacionadas por la herencia). Después de que estudie el capítulo 10, estará familiarizado con las clases, el encapsulamiento, la herencia y el polimorfismo (los aspectos más cruciales de la programación orientada a objetos).

RESUMEN

- La reutilización de software reduce el tiempo de desarrollo de los programas.
- La superclase directa de una subclase es la superclase de la cual hereda (se especifica mediante la palabra clave `extends` en la primera línea de una declaración de clase). Una superclase indirecta de una subclase se encuentra a dos o más niveles hacia arriba de distancia de esa subclase, en la jerarquía de clases.
- En la herencia simple, una clase se deriva de una superclase directa. En la herencia múltiple, una clase se deriva de más de una superclase directa. Java no cuenta con soporte para la herencia múltiple.
- Una subclase puede declarar a sus propios miembros (variables de instancia y métodos), por lo que una subclase es a menudo más grande que su superclase.
- Una subclase es más específica que su superclase y representa a un grupo más pequeño de objetos.
- Todo objeto de una subclase es también un objeto de la superclase de esa clase. Sin embargo, un objeto de una superclase no es un objeto de las subclases de esa clase.
- Una relación "es un" representa la herencia. En este tipo de relación, el objeto de una subclase puede tratarse también como un objeto de su superclase.
- Una relación "tiene un" representa la composición. En este tipo de relación, el objeto de una clase contiene referencias a objetos de otras clases.
- Una subclase no puede utilizar ni heredar los miembros `private` de su superclase; si se permitiera esto se quebrantaría el encapsulamiento de la superclase. Sin embargo, una subclase puede heredar los miembros `public` y `protected` de su superclase.
- Cuando el método de una superclase es inapropiado para una subclase, ese método puede sobrescribirse en la subclase mediante una implementación apropiada.
- Las relaciones de herencia simple forman estructuras jerárquicas tipo árbol; una superclase existe en una relación jerárquica con sus subclases.
- Los miembros `public` de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o a un objeto de una de las subclases de esa superclase.
- Los miembros `private` de una superclase son accesibles solamente dentro de la declaración de esa superclase.
- Los miembros `protected` de una superclase tienen un nivel intermedio de protección entre el acceso `public` y el acceso `private`. Los miembros `protected` de una superclase pueden ser utilizados por los miembros de esa superclase, y heredados por los miembros de cualquier clase derivada de esa superclase.

http://librosysolucionarios.net

- Cuando el método de una subclase sobrescribe al método de una superclase, el método de la superclase puede utilizarse desde la subclase si se antepone al nombre del método de la superclase la palabra clave `super` y un punto (.).
- Al declarar las variables de instancia como `private` y proporcionar al mismo tiempo métodos no `private` para manipular y realizar la comprobación de validez, se refuerza la buena ingeniería de software.
- El método `toString` no toma argumentos y devuelve un `String`. El método `toString` original de la clase `Object` sirve para guardar lugar, y generalmente se sobrescribe en una subclase.
- Al concatenar una cadena con cualquier objeto se produce una llamada implícita al método `toString` de ese objeto, para obtener una representación de cadena de un objeto, y después se concatenan las cadenas.

TERMINOLOGÍA

abstracción
biblioteca de clases
composición
constructor de una subclase
constructor de una superclase
constructor sin argumentos de una superclase
constructor sobrecargado
control de acceso
diagrama jerárquico
`extends`
finalizador de una subclase
finalizador de una superclase
herencia
herencia múltiple
herencia simple
invocar al constructor de una superclase
invocar al método sobrescrito de una superclase
ISV (fabricante independiente de software)
jerarquía de clases
jerarquía de herencia
método heredado
miembro heredado
miembro `private` de una superclase
miembro `protected` de una superclase
miembro `public` de una superclase
`Object`, clase
objeto de una subclase
objeto de una superclase
ocultamiento de información
POO (programación orientada a objetos)
`protected`, palabra clave
recolección de basura
referencia a un objeto de una subclase
referencia a un objeto de una superclase
relación *es un*
relación jerárquica
relación *tiene un*
reutilización de software
sobrescribir el método de una superclase
subclase
`super`
superclase
superclase directa
superclase indirecta
`toString`, método de la clase `Object`
variable de instancia heredada

EJERCICIOS DE AUTOEVALUACIÓN

9.1 Complete las siguientes oraciones:

a) _____ es una forma de reutilización de software, en la que nuevas clases adquieren los datos y comportamientos de las clases existentes, y se mejoran con nuevas capacidades.
b) Los miembros _____ de una superclase pueden utilizarse solamente en la declaración de la superclase o pueden ser heredados por las subclases.
c) En una relación _____, un objeto de una subclase puede ser tratado también como un objeto de su superclase.
d) En una relación _____, el objeto de una clase tiene referencias a objetos de otras clases como miembros.
e) En la herencia simple, una clase existe en una relación _____ con sus subclases.
f) Los miembros _____ de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o a un objeto de una de sus subclases.
g) Los miembros de acceso `protected` de una superclase tienen un nivel de protección entre los miembros de acceso _____ y los de acceso _____.
h) Cuando se crea la instancia de un objeto de una subclase, el _____ de una superclase se llama en forma implícita o explícita.
i) Los constructores de una subclase pueden llamar a los constructores de la superclase mediante la palabra clave _____.

9.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Es posible tratar a los objetos de una superclase y a los objetos de una subclase de manera similar.
b) Los constructores de la superclase no son heredados por las subclases.
c) Una relación "tiene un" se implementa mediante la herencia.
d) La clase `Auto` tiene una relación "es un" con su `VolanteDireccion` y sus `Frenos`.
e) La herencia fomenta la reutilización de software comprobado, de alta calidad.

f) Cuando una subclase redefine al método de una superclase utilizando la misma firma, se dice que la subclase sobrecarga a ese método de la superclase.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

9.1 a) Herencia. b) `protected`. c) "Es un" o de herencia. d) "Tiene-un", o composición o agregación. e) Jerárquica. f) `public`. g) `public, private`. h) Constructor. i) `super`.

9.2 a) Verdadero. b) Verdadero. c) Falso. Una relación "tiene un" se implementa mediante la composición. Una relación "es un" se implementa mediante la herencia. d) Falso. Éste es un ejemplo de una relación "es un". La clase `Auto` tiene una relación "es un" con la clase `Vehiculo`. e) Verdadero. f) Falso. Esto se conoce como sobrescritura, no sobrecarga.

EJERCICIOS

9.3 Muchos programas escritos con herencia podrían escribirse mediante la composición, y viceversa. Vuelva a escribir las clases `Circulo4` (figura 9.13) y `Cilindro` (figura 9.15) de la jerarquía `Punto3/Circulo4/Cilindro`, para usar la composición en vez de la herencia. Una vez que haga esto, valore los méritos relativos de las dos metodologías para los problemas del `Punto3`, `Circulo4` y `Cilindro`, así como también para los programas orientados a objetos en general. ¿Cuál metodología es más natural? ¿Por qué?

9.4 Algunos programadores prefieren no utilizar acceso `protected`, ya que creen que quebranta el encapsulamiento de la superclase. Hable sobre los méritos relativos de utilizar acceso `protected`, en comparación con utilizar el acceso `private` en las superclases.

9.5 Vuelva a escribir el ejemplo práctico de la sección 9.5, como una jerarquía `Punto/Cuadrado/Cubo`. Haga esto de dos formas: una mediante la herencia y una mediante la composición.

9.6 Escriba una jerarquía de herencia para las clases `Cuadrilatero`, `Trapezoide`, `Paralelogramo`, `Rectangulo` y `Cuadrado`. Use `Cuadrilatero` como la superclase de la jerarquía. Haga la jerarquía con todos los niveles que sea posible. Especifique las variables de instancia y los métodos para cada clase. Los datos `private` de `Cuadrilatero` deben ser los pares de coordenadas *x*-*y* para las cuatro esquinas del `Cuadrilatero`. Escriba un programa para instanciar objetos de sus clases y que muestre el área de cada objeto (excepto `Cuadrilatero`).

9.7 Anote todas las figuras en las que pueda pensar (tanto bidimensionales como tridimensionales) y ordene esas figuras en una jerarquía de figuras. Su jerarquía debe tener la clase `Figura` en la parte superior. Las clases `FiguraBidimensional` y `FiguraTridimensional` deben extender a `Figura`. Una vez que haya desarrollado la jerarquía, declare a cada una de las clases en ella. Utilizaremos esta jerarquía en los ejercicios para procesar todas las figuras como objetos de la superclase `Figura`.

9.8 Cree las clases que se encuentran en la jerarquía de herencia de la figura 9.20. Un `Empleado` debe tener un primer nombre, apellido paterno y número de seguro social. Además, un `EmpleadoAsalariado` debe tener un salario semanal; un `EmpleadoPorHoras` debe tiene un sueldo y el número de horas trabajadas; un `EmpleadoPorComision` debe tener una tasa de comisiones y ventas brutas; y un `EmpleadoBaseMasComision` debe tener un salario base. Cada clase debe tener los constructores apropiados, los métodos *establecer* y *obtener*. Escriba un programa que cree instancias de objetos de cada una de estas clases, y que muestre toda la información asociada con cada objeto (incluyendo la información heredada).

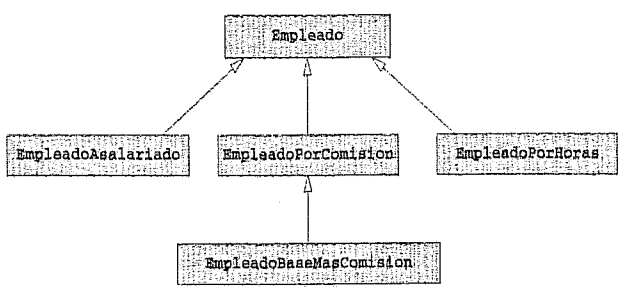
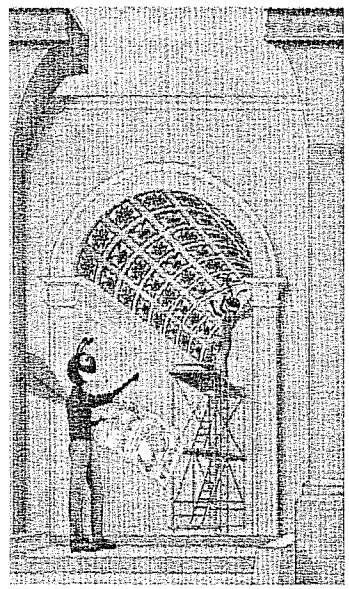


Figura 9.20 Jerarquía de herencia de `Empleado`.

http://librosysolucionarios.net

10

Programación orientada a objetos: Polimorfismo



Objetivos

- Comprender el concepto de polimorfismo.
- Aprender a utilizar métodos sobrescritos para llevar a cabo el polimorfismo.
- Distinguir entre clases abstractas y concretas.
- Aprender a declarar métodos `abstract` para crear clases abstractas.
- Apreciar la manera en que el polimorfismo hace que los sistemas puedan extenderse y mantenerse.
- Determinar el tipo de un objeto en tiempo de ejecución.

Un anillo para gobernarlos a todos, un anillo para encontrarlos,
Un anillo para traerlos a todos y en la oscuridad enlazarlos.
John Ronald Reuel Tolkien

Las proposiciones generales no deciden casos concretos.
Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío. Incluso sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.
Alfred North Whitehead

Plan general

10.1 Introducción
10.2 Relaciones entre los objetos en una jerarquía de herencia
10.2.1 Invocación de los métodos de superclases desde objetos de subclases
10.2.2 Uso de referencias a superclases con variables tipo subclase
10.2.3 Llamadas a métodos de subclases mediante variables tipo superclase
10.3 Ejemplos de polimorfismo
10.4 Clases y métodos abstractos
10.5 Ejemplo práctico: Herencia de interfaz y de implementación
10.6 Métodos y clases `final`
10.7 Ejemplo práctico: Sistema de nómina utilizando polimorfismo
10.8 Ejemplo práctico: Creación y uso de interfaces
10.9 Clases anidadas
10.10 Clases de tipo de envoltura para los tipos primitivos
10.11 (Ejemplo práctico opcional) Acerca de los objetos: Cómo incorporar la herencia a la simulación del elevador
10.12 (Opcional) Descubrimiento de patrones de diseño: Introducción a los patrones de diseño de creación, estructura y comportamiento
10.12.1 Patrones de diseño de creación
10.12.2 Patrones de diseño de estructura
10.12.3 Patrones de diseño de comportamiento
10.12.4 Conclusión
10.12.5 Recursos en Internet y World Wide Web
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

10.1 Introducción

En el capítulo 8 hablamos acerca de la programación orientada a objetos y sus tecnologías componentes: clases, objetos, encapsulamiento y abstracción de datos. En el capítulo 9 nos enfocamos en una tecnología clave de la POO (programación orientada a objetos): la herencia. Ahora continuaremos nuestro estudio de la POO explicando y demostrando el *polimorfismo* con las jerarquías de herencia. El polimorfismo nos permite "programar en forma general", en vez de "programar en forma específica". En especial, el polimorfismo nos permite escribir programas que procesen objetos de clases que formen parte de la misma jerarquía de clases, como si todos fueran objetos de sus superclases. Como veremos pronto, el polimorfismo funciona mediante las referencias a la superclase.

En este capítulo presentamos varias partes clave. Comenzaremos con una secuencia de ejemplos pequeños y precisos que demostrarán las relaciones entre las clases en una jerarquía. En el ejemplo inicial proporcionamos nuestra primera introducción al polimorfismo y demostramos las maneras permitidas de asignar objetos de superclases y subclases a las referencias a superclases y subclases. Los demás ejemplos en esta secuencia muestran los errores que ocurren cuando un programa trata de realizar asignaciones que no están permitidas. Después presentaremos un ejemplo práctico en el que volvemos a utilizar la jerarquía `Punto3/Circulo/Cilindro` de la sección 9.5. En este ejemplo práctico declaramos una "interfaz" común (es decir, conjunto de funcionalidad) para todas las clases en la jerarquía. Esta funcionalidad común entre las formas se declara en una supuesta *clase abstracta* llamada `Forma`, de la cual la clase `Punto` hereda directamente y las clases `Circulo` y `Cilindro` heredan indirectamente.

http://librosysolucionarios.net

Después presentaremos una jerarquía de clases `Empleado` más "natural". Desarrollaremos un sistema de nómina simple para esta jerarquía, en la que cada empleado tiene un método de ingresos que calcula el pago semanal. Estos métodos de ingresos varían según el tipo de empleado; a un `EmpleadoAsalariado` se le paga un salario semanal fijo, sin importar el número de horas trabajadas, a un `EmpleadoPorHoras` se le paga por hora y recibe pago por tiempo extra, un `EmpleadoPorComision` recibe un porcentaje de sus ventas y un `EmpleadoBaseMasComision` recibe un salario base más un porcentaje de sus ventas. Mostraremos cómo procesar cada uno de los empleados "en general", invocando al método `ingresos` desde una referencia a la superclase.

En ocasiones, cuando se lleva a cabo el procesamiento del polimorfismo es necesario programar "en forma específica". Nuestro ejemplo práctico con `Empleado` demuestra que un programa puede determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto de manera acorde. En el ejemplo práctico utilizamos estas capacidades para determinar si cierto objeto empleado *es un* `EmpleadoBaseMasComision`; si es así, otorgamos a ese empleado una bonificación del 10% sobre su salario base.

Con el polimorfismo es posible diseñar e implementar sistemas que puedan extenderse fácilmente. Pueden agregarse nuevas clases con pocas modificaciones (o si acaso ninguna) a las porciones genéricas del programa, siempre y cuando esas clases formen parte de la jerarquía de herencia que el programa procese en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a nuevas clases son aquellos componentes del programa que requieren de un conocimiento directo de las nuevas clases que el programador agregue a la jerarquía.

El capítulo continúa con una introducción a las interfaces en Java. Una interfaz describe a un conjunto de métodos que pueden llamarse en un objeto. Los programadores pueden declarar clases que *implementen* a una o más interfaces. Cada método de interfaz debe declararse en la clase que implementa a esa interfaz. Una vez que una clase implementa a una interfaz, todos los objetos de esa clase tienen la relación *es un* con el tipo de la interfaz, y se garantiza que todos los objetos de la clase proporcionarán la funcionalidad descrita por la interfaz. Como veremos, esto permite capacidades de polimorfismo similares a las que se demuestran con la herencia en la primera parte de este capítulo.

Después de presentar las interfaces, hablaremos más acerca de los detalles del manejo de eventos, en donde se utilizan interfaces para permitir a los componentes de la GUI interactuar con objetos manejadores de eventos. Demostraremos estos conceptos en el contexto de nuestra primera aplicación que se ejecutará en su propia ventana. En estos ejemplos también demostraremos una nueva forma de clase conocida como *clase interna*. Las clases internas se declaran completamente en el interior de otras clases.

10.2 Relaciones entre los objetos en una jerarquía de herencia

En la sección 9.4 se creó una jerarquía de clases *punto/circulo*, en donde la clase `Circulo` heredó de la clase `Punto`. Los ejemplos en esa sección manipularon objetos `Punto` y `Circulo` mediante el uso de referencias a esos objetos, para invocar a sus métodos. Ahora examinaremos las relaciones entre las clases de una jerarquía. Los siguientes ejemplos demuestran cómo se les pueden asignar referencias a objetos de superclases y subclases a las variables de superclases y de subclases, y cómo pueden usarse esas referencias para invocar métodos que manipulen a esos objetos.

En la sección 10.2.1 asignaremos la referencia a una superclase al objeto de una subclase. Después mostraremos cómo al invocar un método a través de la referencia a la superclase se invoca a la funcionalidad de la subclase; el tipo del objeto referenciado determina a cuál método se llama. En la sección 10.2.2 asignaremos la referencia a una subclase al objeto de una superclase, lo cual producirá un error de compilación. Hablaremos sobre este mensaje de error e investigaremos por qué el compilador no permite dicha asignación. En la sección 10.2.3 asignaremos la referencia a una superclase al objeto de una subclase, para demostrar que la referencia a una superclase puede utilizarse para invocar sólo a la funcionalidad de la superclase; cuando tratamos de invocar a los métodos de una subclase a través de la referencia a la superclase, se producen errores de compilación.

Un concepto clave demostrado por estos ejemplos es que un objeto de una subclase puede tratarse como un objeto de su superclase. Esto permite varias manipulaciones interesantes. Por ejemplo, un programa podría crear un arreglo de referencias a la superclase, que hagan referencia a los objetos de muchos tipos de subclase. Esto se permite, a pesar del hecho de que los objetos de las subclases son de distintos tipos, ya que cada

objeto de una subclase *es un* objeto de su superclase. Sin embargo, el objeto de una superclase no es un objeto de ninguna de sus subclases. Por ejemplo, un `Punto` no es un `Circulo` en la jerarquía que se muestra en la sección 9.4 (un `Punto` no tiene una variable de instancia llamada `radio` ni los métodos `establecerRadio`, `obtenerRadio` y `obtenerArea`). La relación *es un* se aplica solamente desde una subclase hacia sus superclases directa e indirecta.

10.2.1 Invocación de los métodos de superclases desde objetos de subclases

El ejemplo de la figura 10.1 demuestra tres maneras de utilizar variables de tipo de superclase y subclase para almacenar referencias a objetos de la superclase y la subclase. Las primeras dos son simples; asignamos a una variable de tipo de superclase la referencia a una superclase, y asignamos a una variable tipo subclase la referencia a una subclase. Después demostramos la relación entre subclases y superclases (es decir, la relación *es un*) asignando la referencia a una subclase a una variable de la superclase. (*Nota:* En este programa utilizamos las clases `Punto3` y `Circulo4` de las figuras 9.12 y 9.13.)

```
1  // Fig. 10.1: PruebaRelacionJerarquia1.java
2  // Asignación de referencias de superclase y subclase a variables de tipo
3  // superclase y subclase.
4  import javax.swing.JOptionPane;
5
6  public class PruebaRelacionJerarquia1 {
7
8     public static void main( String[] args )
9     {
10       // asignar a la variable de tipo de superclase, la referencia a la superclase
11       Punto3 punto = new Punto3( 30, 50 );
12
13       // asignar a la referencia de la subclase, la variable tipo subclase
14       Circulo4 circulo = new Circulo4( 120, 89, 2.7 );
15
16       // invocar a toString en objeto de superclase usando variable de superclase
17       String salida = "Llamar a toString de Punto3 con la referencia a la" +
18          " superclase apuntando al objeto de la superclase: \n" + punto.toString();
19
20       // invocar a toString en objeto de subclase usando variable de subclase
21       salida += "\n\nLlamar a toString de Circulo4 con la referencia a la" +
22          " subclase apuntando al objeto de la subclase: \n" +
23          circulo.toString();
24
25       // invocar a toString en objeto de subclase utilizando variable de
26          superclase
27       Punto3 refPunto = circulo;
28       salida += "\n\nLlamar a toString de Circulo4 con la referencia a la" +
29          " superclase apuntando al objeto de la subclase: \n" + refPunto.toString();
30
31       JOptionPane.showMessageDialog( null, salida );  // mostrar la salida
32
```

Figura 10.1 Asignación de referencias a superclases y subclases, a los objetos de superclases y subclases. (Parte 1 de 2.)


```
33    } // fin de main
34
35 } // fin de la clase PruebaRelacionJerarquia1
```

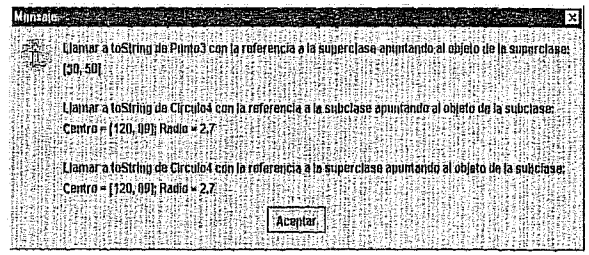


Figura 10.1 Asignación de referencias a superclases y subclases, a los objetos de superclases y subclases. (Parte 2 de 2.)

En la línea 11 de la figura se crea un objeto `Punto3` y se le asigna su referencia a la variable `punto` de `Punto3`. En la línea 14 se crea un objeto `Circulo4` y se le asigna su referencia a la variable `circulo` de `Circulo4`. En las líneas 17 y 18 se usa la referencia `punto` para invocar a `toString` en el objeto `Punto3`, el cual llama a la versión de `toString` de la superclase `Punto3`. De manera similar, en las líneas 21 y 22 se utiliza `circulo` para invocar a `toString` en el objeto `Circulo4`. Esto invoca a la versión de `toString` de la subclase `Circulo4`. Después, en la línea 25 se asigna la referencia del objeto `circulo` de la subclase a la variable `refPunto` de tipo superclase, que en las líneas 26 y 27 se utiliza para invocar al método `toString`. Una variable de tipo superclase que contiene una referencia al objeto de la subclase llama al método de la subclase; por lo tanto, `refPunto.toString()` en realidad llama al método `toString()` de la clase `Circulo4`. El compilador de Java permite este "cruzamiento", ya que un objeto de una subclase *es un* objeto de su superclase. Para cualquier llamada a un método no `static`, el tipo de la variable utilizada para llamar al método determina cuáles métodos pueden llamarse. El tipo del objeto al que la variable hace referencia determina los métodos que se utilizarán para responder a esa llamada a un método.

10.2.2 Uso de referencias a superclases con variables tipo subclase

En la sección 10.2.1 asignamos la referencia del objeto de una subclase a una variable de tipo de superclase y explicamos que el compilador de Java permite esta asignación, ya que el objeto de una subclase *es un* objeto de la superclase. Ahora utilizaremos una metodología opuesta en la figura 10.2, en donde trataremos de asignar la referencia a un objeto de la superclase a la variable tipo subclase. (*Nota:* En este programa utilizamos las clases `Punto3` y `Circulo4` de las figuras 9.12 y 9.13.) En la línea 8 se crea un objeto `Punto3`, y en la línea 9 se declara una variable `Circulo4`. En la línea 12 se intenta asignar la referencia del objeto `punto` de la superclase a `circulo`, pero el compilador de Java genera un error. El compilador evita esta asignación, ya que un `Punto3` no es un `Circulo4` (la relación *es un* se aplica sólo entre la subclase y sus superclases).

```
1  // Fig. 10.2: PruebaRelacionJerarquia2.java
2  // Intento de asignar la referencia de una superclase a una variable de tipo subclase.
3
4  public class PruebaRelacionJerarquia2 {
5
6     public static void main( String[] args )
7     {
```

Figura 10.2 Asignación de una referencia de subclase a un objeto de superclase. (Parte 1 de 2.)

http://librosysolucionarios.net

```
8         Punto3 punto = new Punto3( 30, 50 );
9         Circulo4 circulo;  // variable de tipo subclase
10
11        // asignar la referencia de la superclase a la variable de tipo subclase
12        circulo = punto;  // Error: un Punto3 no es un Circulo4
13     }
14
15  } // fin de la clase PruebaRelacionJerarquia2
```

```
PruebaRelacionJerarquia2.java:12: incompatible types
found   : Punto3
required: Circulo4
      circulo = punto;  // Error: un Punto3 no es un Circulo4
                ^
1 error
```

Figura 10.2 Asignación de una referencia de subclase a un objeto de superclase. (Parte 2 de 2.)

Resulta que el compilador de Java permite esta asignación si convertimos explícitamente la referencia de la superclase al tipo de la subclase (una técnica que describiremos con más detalle en la sección 10.7. Tal vez se pregunte qué caso tiene realizar esa asignación. En programas que procesan objetos de superclases y subclases mediante el uso de referencias a la superclase, sólo los métodos de la superclase pueden ser invocados mediante las referencias a la superclase. La conversión de referencias de la superclase a referencias de la subclase (lo que también se conoce como *conversión descendente* o "*downcasting*") permite a un programa invocar la funcionalidad de la subclase para realizar operaciones específicas de la subclase en sus objetos.



Error común de programación 10.1

Asignar la referencia de un objeto de la superclase a una variable tipo subclase (sin una conversión explícita) es un error de compilación.



Observación de ingeniería de software 10.1

Si la referencia del objeto de una subclase se asigna a una variable de una de sus superclases directas o indirectas, es aceptable convertir esa referencia a la superclase de vuelta a una referencia del tipo de la subclase. De hecho, esto debe hacerse para invocar métodos de la subclase que no aparezcan en la superclase.

10.2.3 Llamadas a métodos de subclases mediante variables tipo superclase

Desde la referencia a una subclase, el compilador nos permite invocar a todos los métodos de la subclase. Por lo tanto, si pudiera asignarse una referencia a la subclase a un objeto de la superclase, y si se hiciera el intento de acceder a un método único de la subclase, sería muy probable que se produjeran errores. En la sección 10.2.2 vimos que tratar de utilizar una variable de subclase para hacer referencia a un objeto de superclase es un error de compilación.

En la figura 10.3 se trata de llamar a un método de la subclase con una referencia a la superclase. (*Nota:* En este ejemplo utilizamos las clases `Punto3` y `Circulo4` de las figuras 9.12 y 9.13.) En la línea 9 se declara la variable `Punto3` llamada `punto`. En la línea 10 se declara la variable `Circulo4` llamada `circulo` y se crea un objeto `Circulo4`. En la línea 12 se asigna la referencia del objeto de la subclase `Circulo4` a la variable `punto` de la superclase `Punto3`. Recuerde que en la sección 10.2.1 vimos que el compilador de Java permite esto, ya que un `Circulo4` *es un* `Punto3`. En las líneas 16 a 20 se invocan los métodos de la superclase `obtenerX`, `obtenerY`, `establecerX`, `establecerY` y `toString` mediante la referencia a la superclase. Sabemos que `punto` contiene una referencia a un objeto `Circulo4`, por lo que en las líneas 24 a 28 tratamos de invocar a los métodos `obtenerRadio`, `establecerRadio`, `obtenerDiametro`, `obtenerCircunferencia` y `obtenerArea` de `Circulo4`. El compilador de Java genera errores en cada una de estas líneas, ya que éstos no son métodos de la superclase `Punto3`. Aunque el método que se llama en realidad depende del tipo del objeto en tiempo de ejecución, puede utilizarse una variable para invocar sólo a aquellos métodos que sean miembros del tipo de esa variable, lo cual puede comprobar el compilador en tiempo

de compilación. (En este caso, utilizando una variable `Punto3` sólo podemos invocar a los métodos `obtenerX`, `obtenerY`, `establecerX`, `establecerY` y `toString`.)

Resumen de las asignaciones permitidas entre las variables de la superclase y la subclase

A pesar del hecho de que el objeto de una subclase también *es un* objeto de la superclase, los objetos de la subclase y los de la superclase son ciertamente diferentes. Como lo hemos dicho anteriormente, los objetos de la subclase pueden tratarse como si fueran objetos de la superclase. Ésta es una relación lógica, ya que la subclase contiene a todos los miembros de la superclase, pero la subclase puede tener miembros adicionales exclusivos. Por esta razón, asignar una referencia de la superclase a una variable tipo subclase no se permite sin una conversión explícita de tipos; dicha asignación dejaría indefinidos a los miembros exclusivos de la subclase en el objeto de la superclase.

Hemos visto cuatro maneras de asignar referencias de superclase y subclase a variables de tipos superclase y subclase:

1. La asignación de una referencia de superclase a una variable tipo superclase es simple y directa.
2. La asignación de una referencia de subclase a una variable tipo subclase es simple y directa.
3. La asignación de la referencia de un objeto de la subclase a una variable tipo superclase es segura, ya que el objeto de la subclase *es un* objeto de su superclase. Sin embargo, esta referencia puede utilizarse para invocar sólo a los métodos de la superclase. Si este código hace referencia a los miembros exclusivos de la subclase mediante la variable de superclase, el compilador reporta errores.

```
1  // Fig. 10.3: PruebaRelacionJerarquia3.java
2  // Intento de invocar a los métodos miembros sólo de la subclase
3  // a través de una referencia a la superclase.
4
5  public class PruebaRelacionJerarquia3 {
6
7     public static void main( String[] args )
8     {
9        Punto3 punto;
10       Circulo4 circulo = new Circulo4( 120, 89, 2.7 );
11
12       punto = circulo;  // apuntar referencia de superclase al objeto de subclase
13
14       // invocar métodos de superclase (Punto3) en objeto de subclase
15       // (Circulo4) a través de una referencia a la superclase
16       int x = punto.obtenerX();
17       int y = punto.obtenerY();
18       punto.establecerX( 10 );
19       punto.establecerY( 20 );
20       punto.toString();
21
22       // intentar invocar a los métodos sólo de la subclase (Circulo4) en el
23       // objeto de la subclase, a través de una referencia a la superclase (Punto3)
24       double radio = punto.obtenerRadio();
25       punto.establecerRadio( 33.33 );
26       double diametro = punto.obtenerDiametro();
27       double circunferencia = punto.obtenerCircunferencia();
28       double area = punto.obtenerArea();
29
30    } // fin de main
31
32 } // fin de la clase PruebaRelacionJerarquia3
```

Figura 10.3 Intento de invocar métodos exclusivos de la subclase mediante una referencia a la superclase. (Parte 1 de 2.)

http://librosysolucionarios.net

```
PruebaRelacionJerarquia3.java:24: cannot resolve symbol
symbol  : method obtenerRadio ()
location: class Punto3
      double radio = punto.obtenerRadio();
                           ^
PruebaRelacionJerarquia3.java:25: cannot resolve symbol
symbol  : method establecerRadio (double)
location: class Punto3
      punto.establecerRadio( 33.33 );
            ^
PruebaRelacionJerarquia3.java:26: cannot resolve symbol
symbol  : method obtenerDiametro ()
location: class Punto3
      double diametro = punto.obtenerDiametro();
                              ^
PruebaRelacionJerarquia3.java:27: cannot resolve symbol
symbol  : method obtenerCircunferencia ()
location: class Punto3
      double circunferencia = punto.obtenerCircunferencia();
                                    ^
PruebaRelacionJerarquia3.java:28: cannot resolve symbol
symbol  : method obtenerArea ()
location: class Punto3
      double area = punto.obtenerArea();
                          ^
5 errors
```

Figura 10.3 Intento de invocar métodos exclusivos de la subclase mediante una referencia a la superclase. (Parte 2 de 2.)

4. Si se intenta asignar la referencia de un objeto de superclase a una variable tipo subclase se produce un error de compilación. Para evitar este error, la referencia a la superclase debe convertirse en el tipo subclase de manera explícita. En tiempo de ejecución, si el objeto al cual apunta la referencia no es un objeto de la subclase, se producirá una excepción. En la sección 10.7 demostraremos cómo asegurarnos que dicha conversión se lleve a cabo solamente si el objeto es un objeto de la subclase.



Error común de programación 10.2

Después de apuntar la referencia de una superclase a un objeto de la subclase, tratar de hacer referencia a los miembros exclusivos de la subclase con la referencia a la superclase es un error de compilación.

10.3 Ejemplos de polimorfismo

Considere el siguiente ejemplo de polimorfismo. Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero`, entonces un objeto `Rectangulo` es una versión más específica de un objeto `Cuadrilatero`. Cualquier operación (por ejemplo, calcular el perímetro o el área) que pueda llevarse a cabo en un objeto `Cuadrilatero` también puede realizarse en un objeto `Rectangulo`. Estas operaciones pueden llevarse a cabo también en otros objetos `Cuadrilatero`, como los objetos `Cuadrado`, `Paralelogramo` y `Trapezoide`. Cuando un programa invoca a un método a través de una variable de la superclase, Java selecciona, mediante el polimorfismo, la versión correcta del método de la subclase a llamar, con base en el tipo de la referencia almacenada en la variable de la superclase.

Como otro ejemplo, suponga que diseñaremos un videojuego que manipule objetos de muchos tipos distintos, incluyendo objetos de las clases `Marciano`, `Venusiano`, `Plutoniano`, `NaveEspacial` y `RayoLaser`. Imagine además que cada una de estas clases hereda de la superclase común llamada `ObjetoEspacial`, la cual contiene el método `dibujar`. Cada subclase implementa este método. Un programa administrador de pantallas mantendría una colección (como un arreglo `SpaceObject`) de referencia a los objetos de las diversas

http://librosysolucionarios.net

clases. Para actualizar la pantalla, el administrador de pantallas enviaría periódicamente a cada objeto el mismo mensaje: `dibujar`. Sin embargo, cada objeto responde en forma única. Por ejemplo, un objeto `Marciano` podría dibujarse a sí mismo en color rojo, con el número apropiado de antenas. Un objeto `NaveEspacial` podría dibujarse a sí mismo como un platillo volador color plateado. Un `RayoLaser` podría dibujarse a sí mismo como un brillante rayo rojo a lo largo de la pantalla. De nuevo, el mismo mensaje (en este caso, `dibujar`) enviado a una variedad de objetos tendría "muchas formas" como resultados; de aquí que se utilice el término *polimorfismo*.

Un administrador de pantallas polimórfico podría utilizar el polimorfismo para facilitar que se agreguen nuevas clases a un sistema, con mínimas modificaciones en el código. Suponga que queremos agregar objetos `Mercuriano` a nuestro videojuego. Para ello debemos crear una clase `Mercuriano` que extienda a `ObjetoEspacial`, pero que proporcione su propia implementación del método `dibujar`. Luego, cuando aparezcan objetos de la clase `Mercuriano` en el contenedor, el programador no tendrá que modificar el código para el administrador de pantallas. Éste invocará al método `dibujar` en todos los objetos del contenedor, sin importar el tipo del objeto, por lo que los nuevos objetos `Mercuriano` simplemente "se integran sin problemas". Por ende y sin necesidad de modificar el sistema (más que para crear e incluir las clases en sí, y para modificar el código que crea nuevos objetos), los programadores pueden utilizar el polimorfismo para incluir tipos adicionales que no se hayan considerado al momento de crear el sistema.

Con el polimorfismo pueden usarse el mismo nombre y firma de los métodos para hacer que ocurran distintas acciones, dependiendo del tipo del objeto sobre el cual se invoca el método. Esto da al programador una enorme capacidad de expresión.



Observación de ingeniería de software 10.2

El polimorfismo permite a los programadores tratar con las generalidades y dejar que el entorno en tiempo de ejecución se encargue de los aspectos específicos. Los programadores pueden ordenar a los objetos que se comporten en formas apropiadas para esos objetos, sin necesidad de conocer sus tipos (siempre y cuando los objetos pertenezcan a la misma jerarquía de herencia).



Observación de ingeniería de software 10.3

El polimorfismo fomenta la extensibilidad: el software que invoca al comportamiento polimórfico es independiente de los tipos de objetos a los que se envían los mensajes. También pueden incorporarse a un sistema nuevos tipos de objetos que puedan responder a las llamadas de los métodos existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que cree instancias de nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

10.4 Clases y métodos abstractos

Cuando pensamos en un tipo de clase, suponemos que los programas crearán objetos de ese tipo. Sin embargo, existen casos en los que es conveniente declarar clases para las cuales el programador no pretende instanciar objetos. Dichas clases se denominan *clases abstractas*. Debido a que éstas se utilizan sólo como superclases en las jerarquías de herencias, nos referimos a esas clases como *superclases abstractas*. Estas clases no pueden utilizarse para instanciar objetos porque, como veremos pronto, las clases abstractas están incompletas. Las subclases deben declarar las "partes faltantes". En las secciones 10.5 y 10.7 crearemos programas con clases abstractas.

El propósito de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases. Las clases que pueden usarse para instanciar objetos se conocen como *clases concretas*. Dichas clases proporcionan implementaciones de todos los métodos que declaran. Podríamos tener una suzperclase abstracta llamada `FiguraBidimensional` y derivar a partir de ella clases concretas como `Cuadrado`, `Circulo` y `Triangulo`. También podríamos tener una superclase abstracta llamada `FiguraTridimensional` y derivar de ella clases concretas como `Cubo`, `Esfera` y `Cilindro`. Las clases abstractas son demasiado genéricas como para crear objetos reales; sólo especifican lo que las subclases tienen en común. Necesitamos ser más específicos antes de poder crear objetos. Por ejemplo, si alguien le dice a usted que "dibuje la figura", ¿qué figura dibujaría? Las clases concretas proporcionan los aspectos específicos que hacen que sea razonable el crear instancias de objetos.

Una jerarquía de herencia no necesita contener clases abstractas. Sin embargo, comúnmente se utilizan jerarquías de clases encabezadas por superclases abstractas para reducir las dependencias de código cliente en

tipos de subclases específicas. En ocasiones las clases abstractas constituyen varios niveles de la jerarquía. Por ejemplo, considere la jerarquía de figuras en la figura 9.3, la cual comienza con la clase abstracta `Figura`. En el siguiente nivel de la jerarquía tenemos otras dos clases abstractas, `FiguraBidimensional` y `FiguraTridimensional`. En el siguiente nivel de la jerarquía se declaran clases concretas para figuras bidimensionales (`Circulo`, `Cuadrado` y `Triangulo`) y para figuras bidimensionales (`Esfera`, `Cubo` y `Tetraedro`).

Para que una clase sea abstracta, se debe declarar con la palabra clave `abstract`. Una clase abstracta normalmente contiene uno o más *métodos abstractos* (los métodos `static` no pueden ser abstractos). Un método abstracto es un método con la palabra clave `abstract` en su declaración, como en:

```
public abstract void dibujar();    // método abstracto
```

Los métodos abstractos no proporcionan implementaciones. Una clase que contiene métodos abstractos debe declararse como una clase abstracta. Cada subclase concreta de una superclase abstracta debe proporcionar implementaciones concretas de los métodos abstractos de la superclase. Los constructores no se heredan, por lo que no pueden declararse como `abstract`.

Observación de ingeniería de software 10.4



Una clase abstracta declara los atributos y comportamientos comunes de las diversas clases en una jerarquía de clases; además, generalmente contiene uno o más métodos abstractos que las subclases deben sobrescribir. Las variables de instancia y los métodos concretos de una clase abstracta se sujetan a las reglas normales de herencia.

Error común de programación 10.3



Al tratar de instanciar un objeto de una clase abstracta se produce un error de compilación.

Error común de programación 10.4



El no poder implementar un método abstracto de la superclase en una subclase es un error de compilación, a menos que la subclase también se declare como `abstract`.

Aunque no podamos crear instancias de objetos de superclases abstractas, *podemos* utilizar superclases abstractas para declarar variables que puedan guardar referencias a objetos de cualquier clase derivada concreta desde esas clases abstractas. Generalmente, los programas utilizan dichas variables para manipular objetos de la subclase mediante el polimorfismo. También podemos usar los nombres de las superclases abstractas para invocar métodos `static` declarados en esas superclases abstractas.

Ahora consideraremos otra aplicación del polimorfismo. Un programa de dibujo necesita mostrar muchas figuras, incluyendo nuevos tipos de figura que el programador agregará al sistema después de escribir el programa de dibujo. Este programa podría necesitar dibujar figuras como `Circulo`, `Triangulo`, `Rectangulo` u otras que se deriven de la superclase abstracta `Figura`. El programa de dibujo utiliza variables `Figura` para administrar los objetos a mostrar. Para dibujar cualquier objeto (sin importar el nivel en el que aparezca la clase de ese objeto en la jerarquía de herencia), el programa de dibujo utiliza una variable de la superclase `Figura` que contiene una referencia al objeto de la subclase, para invocar al método `dibujar` de ese objeto. Este método se declara como `abstract` en la superclase `Figura`; por lo tanto, cada subclase debe implementar el método `dibujar` en una manera que sea específica para esa figura. Cada objeto en la jerarquía de herencia de `Figura` sabe cómo dibujarse a sí mismo. El programa de dibujo no tiene qué preocuparse por el tipo de cada objeto, o si se ha encontrado alguna vez con objetos de un tipo dado.

El polimorfismo es particularmente efectivo para implementar sistemas de software en capas. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico puede operar en forma muy distinta a los demás. Incluso así, los comandos para *leer* o *escribir* datos desde y hacia los dispositivos podrían tener una cierta uniformidad. El mensaje de escritura que se envía a un objeto controlador de dispositivo necesita interpretarse específicamente dentro del contexto de ese controlador de dispositivo, y de cómo ese controlador manipula a los dispositivos de un tipo específico. Sin embargo, la llamada de escritura en sí realmente no es distinta de la escritura en cualquier otro dispositivo en el sistema: colocar cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría utilizar una superclase abstracta para proporcionar una interfaz apropiada para todos los controladores de dispositivos. Después, mediante la herencia de esa superclase abstracta, se forman subclases que operen de manera similar. Los métodos del controlador de dispositivos se declaran como

métodos abstractos en la superclase abstracta. Las implementaciones de estos métodos abstractos se proporcionan en las subclases que corresponden a los tipos específicos de controladores de dispositivos.

Es común en la programación orientada a objetos declarar una *clase iteradora* que pueda recorrer todos los objetos en un contenedor, como un arreglo (capítulo 7) o un objeto `ArrayList` (capítulo 22). Por ejemplo, un programa puede imprimir un arreglo `ArrayList` de objetos creando un objeto iterador, y luego utilizando a ese iterador para obtener el siguiente elemento de la lista cada vez que se llame al iterador. Los iteradores se utilizan comúnmente en la programación polimórfica para recorrer una colección que contiene referencias a objetos de varios niveles de una jerarquía. (En el capítulo 22, Colecciones, presentaremos un análisis extenso de la clase `ArrayList` y los iteradores). Una lista de objetos de la clase `FiguraBidimensional` podría contener objetos de las subclases `Cuadrado`, `Circulo`, `Triangulo` y así, sucesivamente. Si utilizáramos el polimorfismo para llamar al método `dibujar` para cada objeto `FiguraBidimensional`, se dibujaría a cada objeto correctamente en la pantalla.

10.5 Ejemplo práctico: Herencia de interfaz y de implementación

En esta sección analizamos nuevamente la jerarquía `Punto/Circulo/Cilindro` que exploramos en la sección 9.5. En nuestro siguiente ejemplo, la jerarquía comienza con la superclase abstracta `Figura`, la cual declara la "interfaz" para la jerarquía (es decir, el conjunto de métodos que puede invocar un programa en todos los objetos `Figura`). El diagrama de clases de la figura 10.4 empieza con la clase abstracta *`Figura`*. Observe que, en un diagrama de clases, el nombre de una clase abstracta aparece en cursivas.

La clase `Figura` proporciona tres métodos: `obtenerArea`, `obtenerVolumen` y `obtenerNombre`. Además, la clase `Figura` extiende a `Object` y por lo tanto, contiene los 11 métodos de la clase `Object`, entre los que se encuentra `toString`. En el diagrama de la figura 10.5 se muestra cada una de las cuatro clases en la jerarquía por el lado izquierdo, y muestra a los tres métodos de la clase `Figura` y el método `toString` en la parte superior. Para cada clase, el diagrama muestra los resultados deseados de cada método. Observe que la clase `Figura` declara al método `obtenerNombre` como `abstract`, ya que una implementación predeterminada no tiene sentido para este método; no hay suficiente información como para determinar qué cadena debe devolver `obtenerNombre`. Cada subclase sobrescribe a este método para proporcionar las implementaciones apropiadas. Los métodos `obtenerArea` y `obtenerVolumen` tienen cada uno implementaciones predeterminadas, que devuelven `0.0`. La clase `Punto` hereda estas implementaciones (los objetos `Punto` ciertamente tienen un área de `0.0` y un volumen de `0.0`). La clase `Circulo` hereda la implementación predeterminada de `obtenerVolumen` (los objetos `Circulo` ciertamente tienen un volumen de `0.0`) y sobrescribe el método `obtenerArea` para calcular la verdadera área de un círculo. Finalmente, la clase `Cilindro` sobrescribe tanto a `obtenerArea` como a `obtenerVolumen` para realizar los cálculos apropiados para un cilindro.

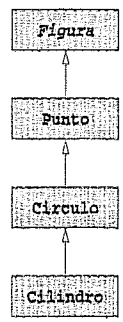


Figura 10.4 Diagrama de clases de la jerarquía `Figura`.

http://librosysolucionarios.net

	obtenerArea	obtenerVolumen	obtenerNombre	toString
Figura	0.0	0.0	abstract	implementación predeterminada de Object
Punto	0.0	0.0	"Punto"	[x,y]
Circulo	πr^2	0.0	"Circulo"	centro=[x,y]; radio=r
Cilindro	$2\pi r^2 + 2\pi rh$	$\pi r^2 h$	"Cilindro"	centro=[x,y]; radio=r; altura=h

Figura 10.5 Interfaz polimórfica para las clases de la jerarquía **Figura**.

Observación de ingeniería de software 10.5

Una subclase puede heredar la "interfaz" o "implementación" de una superclase. Las jerarquías diseñadas para la herencia de implementación suelen tener su funcionalidad en los niveles más bajos de la jerarquía; una superclase especifica a uno o más métodos abstractos que deben declararse para cada una de las clases en la jerarquía, y las subclases individuales sobrescriben a estos métodos para proporcionar implementaciones específicas de la subclase.

La jerarquía en este ejemplo demuestra mecánicamente el poder del polimorfismo. En los ejercicios exploramos una jerarquía de figuras más substancial. La clase abstracta **Figura** (figura 10.6) declara a los métodos **obtenerArea** (líneas 7 a 10) y **obtenerVolumen** (líneas 13 a 16), cada uno de los cuales tiene una implementación que devuelve cero de manera predeterminada, y el método abstracto **obtenerNombre** (línea 19). **Figura** contiene el método abstracto **obtenerNombre**, por lo que debe declararse como clase abstracta (línea 4). Todas las figuras tienen un área y un volumen en este ejemplo, por lo que los métodos **obtenerArea** y **obtenerVolumen** tienen declaraciones concretas en la clase **Figura**. El volumen de las figuras bidimensionales siempre es cero, mientras que las figuras tridimensionales tienen un volumen positivo, distinto de cero. Los programadores pueden sobrescribir los métodos **obtenerArea** y **obtenerVolumen** en las subclases concretas derivadas, mediante las implementaciones apropiadas (vea la figura 10.5). Declaramos el método **obtenerNombre** como un método abstracto, de manera que las subclases que heredan directamente de **Figura** deben implementar a **obtenerNombre** para que puedan ser clases concretas.

```
1  // Fig. 10.6: Figura.java
2  // Declaración de la superclase abstracta Figura.
3
4  public abstract class Figura extends Object {
5
6     // devolver el área de la figura; 0.0 de manera predeterminada
7     public double obtenerArea()
8     {
9        return 0.0;
10    }
11
12    // devolver el volumen de la figura; 0.0 de manera predeterminada
13    public double obtenerVolumen()
14    {
15       return 0.0;
```

Figura 10.6 La clase abstracta **Figura**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
16    }
17
18    // método abstracto, sobrescrito por las subclases
19    public abstract String obtenerNombre();
20
21 } // fin de la clase abstracta Figura
```

Figura 10.6 La clase abstracta **Figura**. (Parte 2 de 2.)

La clase **Punto** (figura 10.7) extiende a la superclase abstracta **Figura** (línea 4) y sobrescribe el método abstracto **obtenerNombre** (líneas 47 a 50), lo cual convierte a **Punto** en una clase concreta. El área y el volumen de un punto son cero, por lo que la clase **Punto** no sobrescribe a los métodos de la superclase **obtenerArea** y **obtenerVolumen**, heredando de esta forma las implementaciones de **Figura** para estos métodos. En las líneas 47 a 50 se implementa el método **obtenerNombre** para devolver la cadena **"Punto"**. Si no implementamos a **obtenerNombre**, la clase **Punto** debe declararse como **abstract**; en caso contrario, se produce un error de compilación.

```
1  // Fig. 10.7: Punto.java
2  // Declaración de la clase Punto que hereda de Figura.
3
4  public class Punto extends Figura {
5     private int x;  // parte x del par de coordenadas
6     private int y;  // parte y del par de coordenadas
7
8     // constructor sin argumentos; valor predeterminado de 0 para x e y
9     public Punto()
10    {
11       // la llamada implícita al constructor de Object ocurre aquí
12    }
13
14    // constructor
15    public Punto( int valorX, int valorY )
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18       x = valorX;  // no hay necesidad de validación
19       y = valorY;  // no hay necesidad de validación
20    }
21
22    // establecer x en el par de coordenadas
23    public void establecerX( int valorX )
24    {
25       x = valorX;  // no hay necesidad de validación
26    }
27
28    // devolver x del par de coordenadas
29    public int obtenerX()
30    {
31       return x;
32    }
33
34    // establecer y en el par de coordenadas
35    public void establecerY( int valorY )
36    {
```

Figura 10.7 La clase **Punto** que extiende a **Figura**. (Parte 1 de 2.)

```
37          y = valorY;  // no hay necesidad de validación
38       }
39
40       // devolver y del par de coordenadas
41       public int obtenerY()
42       {
43          return y;
44       }
45
46       // sobrescribir el método abstracto obtenerNombre para devolver "Punto"
47       public String obtenerNombre()
48       {
49          return "Punto";
50       }
51
52       // sobrescribir a toString para devolver la representación String de Punto
53       public String toString()
54       {
55          return "[" + obtenerX() + ", " + obtenerY() + "]";
56       }
57
58    } // fin de la clase Punto
```

Figura 10.7 La clase **Punto** que extiende a **Figura**. (Parte 2 de 2.)

La clase **Circulo** (figura 10.8) extiende a **Punto** (línea 4) y declara los métodos **establecerRadio** (líneas 21 a 24) y **obtenerRadio** (líneas 27 a 30) para acceder al radio de un círculo. La clase **Circulo** también agrega los métodos **obtenerDiametro** (líneas 33 a 36) y **obtenerCircunferencia** (líneas 39 a 42) para obtener el diámetro y la circunferencia del círculo, respectivamente. Observe que, como un círculo tiene un volumen de cero, **Circulo** no sobrescribe el método **obtenerVolumen** de la superclase. En vez de ello, **Circulo** hereda este método de **Punto** que, a su vez, lo heredó de **Figura**. Sin embargo, un círculo requiere su propio cálculo de área (es decir, πr^2), por lo que **Circulo** sobrescribe el método **obtenerArea** de **Punto** (líneas 45 a 48). El método **obtenerNombre** (líneas 51 a 54) de la clase **Circulo** sobrescribe el método **obtenerNombre** de **Punto**. Si la clase **Circulo** no sobrescribiera este método, heredaría la versión del método **obtenerNombre** correspondiente a **Punto**. En ese caso, el método **obtenerNombre** de **Circulo** hubiera devuelto erróneamente **"Punto"**. Por la misma razón, el método **toString** de **Circulo** (líneas 57 a 60) sobrescribe al método **toString** de **Punto** para devolver información específica de un círculo. Observe que en la línea 59 del método **toString** se invoca al método **toString** de **Punto** para realizar parte del trabajo del método **toString** de **Circulo**; éste es un excelente ejemplo de reutilización de código.

```
1  // Fig. 10.8: Circulo.java
2  // La clase Circulo que hereda de Punto.
3
4  public class Circulo extends Punto {
5     private double radio;  // radio del Circulo
6
7     // constructor sin argumentos; el valor predeterminado de radio es 0.0
8     public Circulo()
9     {
10       // la llamada implícita al constructor de Punto ocurre aquí
11    }
12
```

Figura 10.8 La clase **Circulo** que extiende a **Punto**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
13    // constructor
14    public Circulo( int x, int y, double valorRadio )
15    {
16       super( x, y );  // llamar al constructor de Punto
17       establecerRadio( valorRadio );
18    }
19
20    // establecer el radio
21    public void establecerRadio( double valorRadio )
22    {
23       radio = ( valorRadio < 0.0 ? 0.0 : valorRadio );
24    }
25
26    // devolver el radio
27    public double obtenerRadio()
28    {
29       return radio;
30    }
31
32    // calcular y devolver el diámetro
33    public double obtenerDiametro()
34    {
35       return 2 * obtenerRadio();
36    }
37
38    // calcular y devolver la circunferencia
39    public double obtenerCircunferencia()
40    {
41       return Math.PI * obtenerDiametro();
42    }
43
44    // sobrescribir el método abstracto obtenerArea para devolver área del Circulo
45    public double obtenerArea()
46    {
47       return Math.PI * obtenerRadio() * obtenerRadio();
48    }
49
50    // sobrescribir el método abstracto obtenerNombre para devolver "Circulo"
51    public String obtenerNombre()
52    {
53       return "Circulo";
54    }
55
56    // sobrescribir toString para devolver la representación String de Circulo
57    public String toString()
58    {
59       return "Centro = " + super.toString() + "; Radio = " + obtenerRadio();
60    }
61
62 } // fin de la clase Circulo
```

Figura 10.8 La clase **Circulo** que extiende a **Punto**. (Parte 2 de 2.)

La clase **Cilindro** (figura 10.9) extiende a **Circulo** (línea 4) y sobrescribe a los tres métodos declarados originalmente en la clase **Figura** para proporcionar una funcionalidad específica de **Cilindro**. Como un cilindro tiene distintos cálculos de área y volumen que los de un círculo, **Cilindro** sobrescribe al método

obtenerArea (líneas 33 a 36) para calcular el área superficial del cilindro (es decir, $2\pi r^2 + 2\pi rh$) y sobrescribe al método **obtenerVolumen** (líneas 39 a 42) para calcular el volumen del cilindro ($\pi r^2 h$). Observe que el método **obtenerArea** de **Cilindro** invoca a **obtenerArea** de **Circulo** (línea 35) para realizar parte del cálculo del área; éste es otro excelente ejemplo de reutilización de código. El método **obtenerNombre** (líneas 45 a 48) sobrescribe al método **obtenerNombre** de **Circulo**. Si la clase **Cilindro** no sobrescribiera a este método, heredaría el método **obtenerNombre** de **Circulo**, el cual habría devuelto erróneamente la cadena **"Circulo"**. De manera similar, el método **toString** de **Cilindro** (líneas 51 a 54) sobrescribe al método **toString** de **Circulo** para mostrar la información específica de un cilindro. Una vez más, observe que el método **toString** de **Cilindro** invoca al método **toString** de **Circulo** (línea 53) para obtener la parte del **Cilindro** correspondiente al **Circulo**; otro ejemplo más de reutilización de código.

```
1  // Fig. 10.9: Cilindro.java
2  // La clase Cilindro que hereda de Circulo.
3
4  public class Cilindro extends Circulo {
5     private double altura;  // la altura del Cilindro
6
7     // constructor sin argumentos; el valor predeterminado de altura es 0.0
8     public Cilindro()
9     {
10       // la llamada implícita al constructor de Circulo ocurre aquí
11    }
12
13    // constructor
14    public Cilindro( int x, int y, double radio, double valorAltura )
15    {
16       super( x, y, radio );  // llamar al constructor de Circulo
17       establecerAltura( valorAltura );
18    }
19
20    // establecer la altura del Cilindro
21    public void establecerAltura( double valorAltura )
22    {
23       altura = ( valorAltura < 0.0 ? 0.0 : valorAltura );
24    }
25
26    // obtener la altura del Cilindro
27    public double obtenerAltura()
28    {
29       return altura;
30    }
31
32    // sobrescribir método abstracto obtenerArea para devolver el área de Cilindro
33    public double obtenerArea()
34    {
35       return 2 * super.obtenerArea() + obtenerCircunferencia() * obtenerAltura();
36    }
37
38    // sobrescribir método abstracto obtenerVolumen para devolver valor del
      Cilindro
39    public double obtenerVolumen()
40    {
```

Figura 10.9 La clase **Cilindro** que extiende a **Circulo**. (Parte 1 de 2.)


```
41       return super.obtenerArea() * obtenerAltura();
42    }
43
44    // sobrescribir método abstracto obtenerNombre para devolver "Cilindro"
45    public String obtenerNombre()
46    {
47       return "Cilindro";
48    }
49
50    // sobrescribir toString para devolver representación String del Cilindro
51    public String toString()
52    {
53       return super.toString() + "; Altura = " + obtenerAltura();
54    }
55
56 } // fin de la clase Cilindro
```

Figura 10.9 La clase **Cilindro** que extiende a **Circulo**. (Parte 2 de 2.)

El programa de la figura 10.10 crea un objeto de cada una de las tres clases concretas **Punto**, **Circulo** y **Cilindro**. El programa manipula esos objetos, primero mediante variables del propio tipo de cada objeto (líneas 19 a 22) y después mediante el polimorfismo, utilizando un arreglo de variables **Figura**. En las líneas 14 a 16 se crea un objeto **Punto**, un **Circulo** y un **Cilindro**, y se les asignan sus referencias a las variables **punto**, **circulo** y **cilindro**, respectivamente. Después, en las líneas 19 a 22 se invocan los métodos **obtenerNombre** y **toString** para los objetos, de manera que se obtenga una representación de cadena del nombre y los datos de cada objeto (es decir, el par de coordenadas *x*-*y*, radio y altura, dependiendo del tipo de cada objeto).

```
1  // Fig. 10.10: PruebaHerenciaAbstracta.java
2  // Controlador para la jerarquía figura, punto, circulo, cilindro.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaHerenciaAbstracta {
7
8     public static void main( String args[] )
9     {
10       // establecer el formato de número de punto flotante
11       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
12
13       // crear objetos Punto, Circulo y Cilindro
14       Punto punto = new Punto( 7, 11 );
15       Circulo circulo = new Circulo( 22, 8, 3.5 );
16       Cilindro cilindro = new Cilindro( 20, 30, 3.3, 10.75 );
17
18       // obtener nombre  y representación de cadena de cada objeto
19       String salida = punto.obtenerNombre() + ": " + punto + "\n" +
20          circulo.obtenerNombre() + ": " + circulo + "\n" +
21          cilindro.obtenerNombre() + ": " + cilindro + "\n";
22
23       Figura arregloDeFiguras[] = new Figura[ 3 ];  // crear arreglo de figuras
```

Figura 10.10 El polimorfismo mediante una jerarquía de herencia, encabezada por una superclase abstracta. (Parte 1 de 2.)

http://librosysolucionarios.net

```
24
25          // apuntar arregloDeFiguras[ 0 ] al objeto de la subclase Punto
26          arregloDeFiguras[ 0 ] = punto;
27
28          // apuntar arregloDeFiguras[ 1 ] al objeto de la subclase Circulo
29          arregloDeFiguras[ 1 ] = circulo;
30
31          // apuntar arregloDeFiguras[ 2 ] al objeto de la subclase Cilindro
32          arregloDeFiguras[ 2 ] = cilindro;
33
34          // iterar a través de arregloDeFiguras para obtener nombre,
               representación
35          // de cadena, área y volumen para cada Figura en el arreglo
36          for ( int i = 0; i < arregloDeFiguras.length; i++ ) {
37             salida += "\n\n" + arregloDeFiguras[ i ].obtenerNombre() + ": " +
38                arregloDeFiguras[ i ].toString() + "\nArea = " +
39                dosDigitos.format( arregloDeFiguras[ i ].obtenerArea() ) +
40                "\nVolumen = " +
41                dosDigitos.format( arregloDeFiguras[ i ].obtenerVolumen() );
42          }
43
44          JOptionPane.showMessageDialog( null, salida );  // mostrar resultados
45
46          System.exit( 0 );
47
48       } // fin de main
49
50    } // fin de la clase PruebaHerenciaAbstracta
```

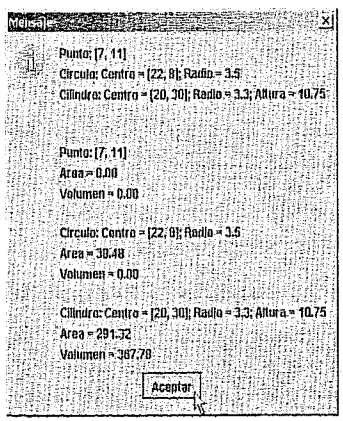


Figura 10.10 El polimorfismo mediante una jerarquía de herencia, encabezada por una superclase abstracta. (Parte 2 de 2.)

En la línea 23 se declara `arregloDeFiguras` y se le asigna un arreglo de tres variables `Figura`. En la línea 26 se asigna al elemento `arregloDeFiguras[ 0 ]` la referencia al objeto `Punto` almacenado en `punto`. En la línea 29 se asigna al elemento `arregloDeFiguras[ 1 ]` la referencia al objeto `Circulo` almacenado en `circulo`. En la línea 32 se asigna al elemento `arregloDeFiguras[ 2 ]` la referencia al objeto `Cilindro` almacenada en `cilindro`. Se permite cada una de estas asignaciones, porque un `Punto`

http://librosysolucionarios.net

es una `Figura`, un `Circulo` *es una* `Figura` y un `Cilindro` *es una* `Figura`. Por lo tanto, podemos asignar las referencias de los objetos `Punto`, `Circulo` y `Cilindro` a las variables de la superclase `Figura`, aun y cuando `Figura` sea una clase abstracta.

Posteriormente, en las líneas 36 a 42 se itera a través del `arregloDeFiguras` y se invoca a los métodos `obtenerNombre`, `toString`, `obtenerArea` y `obtenerVolumen` con `arregloDeFiguras[ i ]`, el cual contiene la referencia a una `Figura`. La salida muestra que ciertamente se invocan los métodos apropiados para cada clase. Primero se muestran la cadena `"Punto"` y las coordenadas del `Punto`; el área y el volumen se muestran como `0.00`. Después, se muestran la cadena `"Circulo"`, las coordenadas del centro del `Circulo` y el radio del mismo; se calcula el área del `Circulo` y el volumen se muestra como `0.00`. Finalmente, se muestran la cadena `"Cilindro"`, las coordenadas del centro de la base del `Cilindro`, su radio y altura; el área y el volumen del `Cilindro` se calculan. Todas las llamadas a `obtenerNombre`, `toString`, `obtenerArea` y `obtenerVolumen` se resuelven en tiempo de ejecución, con base en el tipo del objeto al que se refiere actualmente el elemento `arregloDeFiguras[ i ]`. Este proceso se conoce como *vinculación dinámica* o *vinculación postergada*. En la sección 10.8 veremos de nuevo este ejemplo `Figura/Punto/Circulo/Cilindro`, utilizando la instrucción `interface` de Java para demostrar que también se puede lograr el polimorfismo con las interfaces.

10.6 Métodos y clases `final`

En la sección 6.8 vimos que las variables pueden declararse como `final` para indicar que no pueden modificarse una vez que sean declaradas, y que deben inicializarse cuando sean declaradas. También es posible declarar métodos y clases con el modificador `final`.

Un método que se declara como `final` en una superclase no puede sobrescribirse en una subclase. Los métodos que se declaran como `private` son implícitamente `final`, ya que es imposible sobrescribirlos en una subclase. Los métodos que se declaran como `static` son implícitamente `final`, ya que sólo pueden sobrescribirse los métodos no `static`. Como la declaración de un método `final` nunca puede cambiar, el compilador puede optimizar el programa removiendo las llamadas a los métodos `final` y reemplazándolas con el código expandido de sus declaraciones en la ubicación de cada una de las llamadas a esos métodos; a esta técnica se le conoce como *poner el código en línea*.

Tip de rendimiento 10.1

El compilador puede decidir poner en línea la llamada a un método `final` y lo hará para los métodos `final` pequeños y simples. La puesta en línea no quebranta los principios del encapsulamiento o de ocultamiento de la información (y mejora el rendimiento, ya que elimina la sobrecarga que se produce al realizar la llamada a un método).

Una clase que se declara como `final` no puede ser una superclase (es decir, una clase no puede extender a una clase `final`). Todos los métodos en una clase `final` son implícitamente `final`.

Observación de ingeniería de software 10.6

En la API de Java, la vasta mayoría de clases no se declara como `final`. Esto permite la herencia y el polimorfismo: las características fundamentales de la programación orientada a objetos. Sin embargo, en algunos casos es importante declarar las clases como `final`; generalmente por razones de seguridad o de diseño orientado a objetos.

La clase `String` es un ejemplo de una clase final. Esta clase no puede extenderse, así que los programas que utilizan cadenas pueden depender de la funcionalidad de los objetos `String` según lo especificado en la API de Java. Al hacer la clase `final` también se evita que los programadores creen subclases que podrían pasar por alto las restricciones de seguridad. Para obtener más información sobre las clases y los métodos `final`, visite la página `java.sun.com`.

10.7 Ejemplo práctico: Sistema de nómina utilizando polimorfismo

Ahora emplearemos métodos abstractos y polimorfismo para realizar cálculos de nómina, con base en el tipo de un empleado. Considere el siguiente problema:

Una compañía paga a sus empleados en forma semanal. La compañía tiene cuatro tipos de empleados: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que reciben un sueldo por hora y pago por tiempo extra; empleados por comisión, que reciben un porcentaje de sus ventas y empleados asalariados por comisión, que reciben un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido compensar a los empleados asalariados por comisión, agregando un 10% a sus salarios. La compañía desea implementar una aplicación en Java que realice sus cálculos de nómina en forma polimórfica.

Utilizaremos la clase **Empleado** para representar a un empleado "genérico". Las clases que extienden a **Empleado** son **EmpleadoAsalariado**, **EmpleadoPorComision** y **EmpleadoPorHoras**. La clase **EmpleadoBaseMasComision** (que extiende a **EmpleadoPorComision**) representa el último tipo de empleado. El diagrama de clases de la figura 10.11 muestra la jerarquía de herencia para nuestra aplicación de nómina de empleados. Observe que la clase abstracta ***Empleado*** está en cursivas, según la convención de UML.

Un método **ingresos** se aplica genéricamente a todos los empleados. Pero el cálculo de los ingresos de cada empleado depende de su clase. Por ende, declaramos a **ingresos** como un método abstracto en la superclase **Empleado** (ya que en esta clase no es apropiada ninguna implementación específica) y cada subclase sobrescribe a **ingresos** con una implementación apropiada. Para calcular los ingresos de un empleado, el programa asigna a una variable de la superclase la referencia al objeto de ese empleado, y después invoca al método **ingresos** del empleado. Mantenemos un arreglo de variables **Empleado** que guardan las referencias a cada objeto **Empleado**. El programa itera a través del arreglo y llama al método **ingresos** para cada objeto empleado. Java procesa estas llamadas a los métodos mediante el polimorfismo.

Consideremos ahora la declaración de la clase **Empleado** (figura 10.12). Esta clase incluye un constructor que toma el primer nombre, apellido paterno y número de seguro social como argumentos (líneas 10 a 15); métodos *establecer* que devuelven el primer nombre, apellido paterno y número de seguro social (líneas 24 a 27, 36 a 39 y 48 a 51); métodos *establecer* que establecen el primer nombre, apellido paterno y número de seguro social (líneas 18 a 21, 30 a 33 y 42 a 45); el método **toString** (líneas 54 a 58) que devuelve la representación **String** del **Empleado**; y el método abstracto **ingresos** (línea 61), que se implementará en las subclases. ¿Por qué decidimos declarar el método **ingresos** como un método abstracto? La respuesta es que no tiene sentido proporcionar una implementación de este método en la clase **Empleado**. No podemos calcular los ingresos de un empleado genérico. Primero debemos saber el tipo específico de **Empleado** para determinar el cálculo apropiado de los ingresos. Al declarar este método como **abstract**, indicamos que cada subclase concreta proveerá una implementación apropiada para **ingresos** y que el programa podrá utilizar las variables de la superclase **Empleado** para invocar al método **ingresos** para cualquier tipo de empleado. Observe que el constructor de **Empleado** no valida el número de seguro social en este ejemplo. Generalmente debe proporcionarse dicha validación.

La clase **EmpleadoAsalariado** (figura 10.13) extiende a la clase **Empleado** (línea 4). Esta clase incluye un constructor (líneas 8 a 13) que toma un primer nombre, un apellido paterno, un número de seguro

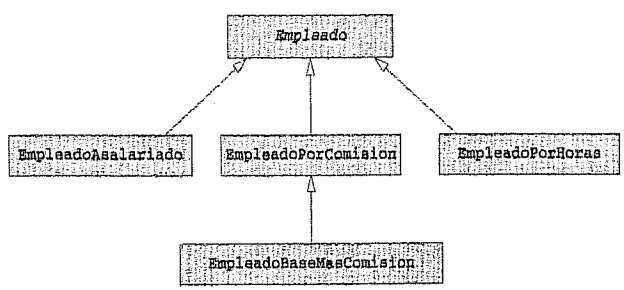


Figura 10.11 Jerarquía de clases para la aplicación polimórfica de una nómina de empleados.

http://librosysolucionarios.net


```
1  // Fig. 10.12: Empleado.java
2  // La superclase abstracta Empleado.
3
4  public abstract class Empleado {
5     private String primerNombre;
6     private String apellidoPaterno;
7     private String numeroSeguroSocial;
8
9     // constructor
10    public Empleado( String nombre, String apellido, String ssn )
11    {
12       primerNombre = nombre;
13       apellidoPaterno = apellido;
14       numeroSeguroSocial = ssn;
15    }
16
17    // establecer el primer nombre
18    public void establecerPrimerNombre( String nombre )
19    {
20       primerNombre = nombre;
21    }
22
23    // devolver el primer nombre
24    public String obtenerPrimerNombre()
25    {
26       return primerNombre;
27    }
28
29    // establecer el apellido
30    public void establecerApellidoPaterno( String apellido )
31    {
32       apellidoPaterno = apellido;
33    }
34
35    // devolver el apellido
36    public String obtenerApellidoPaterno()
37    {
38       return apellidoPaterno;
39    }
40
41    // establecer el número de seguro social
42    public void establecerNumeroSeguroSocial( String numero )
43    {
44       numeroSeguroSocial = numero;  // debería validarse
45    }
46
47    // devolver el número de seguro social
48    public String obtenerNumeroSeguroSocial()
49    {
50       return numeroSeguroSocial;
51    }
52
53    // devolver representación String del objeto Empleado
54    public String toString()
55    {
```

Figura 10.12 La superclase abstracta **Empleado**. (Parte 1 de 2.)

```
56          return obtenerPrimerNombre() + " " + obtenerApellidoPaterno() +
57             "\nnúmero de seguro social: " + obtenerNumeroSeguroSocial();
58       }
59
60       // método abstracto sobrescrito por las subclases
61       public abstract double ingresos();
62
63    } // fin de la clase abstracta Empleado
```

Figura 10.12 La superclase abstracta **Empleado**. (Parte 2 de 2.)

social y un salario semanal como argumentos; un método *establecer* para asignar un nuevo valor a la variable de instancia **salarioSemanal** (líneas 16 a 19); un método *obtener* para devolver el valor de **salarioSemanal** (líneas 22 a 25); un método **ingresos** (líneas 29 a 32) que calcula los ingresos de un **EmpleadoAsalariado**; y un método **toString** (líneas 35 a 38), que devuelve el tipo del empleado, en este caso **"empleado asalariado: "**. El constructor de la clase **EmpleadoAsalariado** pasa el primer nombre, apellido paterno y número de seguro social al constructor de **Empleado** para inicializar los campos heredados de la superclase (línea 11). El método **toString** llama al método **toString** de la superclase (línea 37) para obtener la información específica de la superclase **Empleado** (es decir, primer nombre, apellido paterno y número de seguro social).

```
1  // Fig. 10.13: EmpleadoAsalariado.java
2  // La clase EmpleadoAsalariado que extiende a Empleado.
3
4  public class EmpleadoAsalariado extends Empleado {
5     private double salarioSemanal;
6
7     // constructor
8     public EmpleadoAsalariado( String nombre, String apellido,
9        String numeroSeguroSocial, double salario )
10    {
11       super( nombre, apellido, numeroSeguroSocial );
12       establecerSalarioSemanal( salario );
13    }
14
15    // establecer el salario del empleado asalariado
16    public void establecerSalarioSemanal( double salario )
17    {
18       salarioSemanal = salario < 0.0 ? 0.0 : salario;
19    }
20
21    // devolver el salario del empleado asalariado
22    public double obtenerSalarioSemanal()
23    {
24       return salarioSemanal;
25    }
26
27    // calcular el pago del empleado asalariado;
28    // sobrescribir el método abstracto ingresos en Empleado
29    public double ingresos()
30    {
31       return obtenerSalarioSemanal();
32    }
```

Figura 10.13 La clase **EmpleadoAsalariado** que extiende a **Empleado**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
33
34    // devolver la representación String del objeto EmpleadoAsalariado
35    public String toString()
36    {
37       return "\nempleado asalariado: " + super.toString();
38    }
39
40 } // fin de la clase EmpleadoAsalariado
```

Figura 10.13 La clase **EmpleadoAsalariado** que extiende a **Empleado**. (Parte 2 de 2.)

La clase **EmpleadoPorHoras** (figura 10.14) también extiende a la clase **Empleado** (línea 4). Esta clase incluye un constructor (líneas 9 a 15) que toma un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas como argumentos; métodos *establecer* (líneas 18 a 21 y 30 a 34) para asignar nuevos valores a las variables de instancia **sueldo** y **horas**; métodos *obtener* (líneas 24 a 27 y 37 a 40) para devolver los valores de **sueldo** y **horas**; un método **ingresos** (líneas 44 a 50) que calcula los ingresos de un **EmpleadoPorHoras**; y un método **toString** (líneas 53 a 56), que devuelve el tipo del empleado, en este caso **"empleado por horas: "** e información específica de ese empleado. Observe que, al igual que el constructor de **EmpleadoAsalariado**, el constructor de **EmpleadoPorHoras** también pasa el primer nombre, apellido paterno y número de seguro social al constructor de **Empleado** para inicializar los campos heredados (línea 12). Además, el método **toString** llama al método **toString** de la superclase (línea 55) para obtener la información específica del **Empleado** (es decir, primer nombre, apellido paterno y número de seguro social).

```
1  // Fig. 10.14: EmpleadoPorHoras.java
2  // La clase EmpleadoPorHoras que extiende a Empleado.
3
4  public class EmpleadoPorHoras extends Empleado {
5     private double sueldo;    // sueldo por hora
6     private double horas;     // horas trabajadas en la semana
7
8     // constructor
9     public EmpleadoPorHoras( String nombre, String apellido,
10       String numeroSeguroSocial, double sueldoPorHora, double
         horasTrabajadas )
11    {
12       super( nombre, apellido, numeroSeguroSocial );
13       establecerSueldo( sueldoPorHora );
14       establecerHoras( horasTrabajadas );
15    }
16
17    // establecer sueldo de empleado por horas
18    public void establecerSueldo( double cantidadSueldo )
19    {
20       sueldo = cantidadSueldo < 0.0 ? 0.0 : cantidadSueldo;
21    }
22
23    // devolver sueldo
24    public double obtenerSueldo()
25    {
26       return sueldo;
```

Figura 10.14 La clase **EmpleadoPorHoras** que extiende a **Empleado**. (Parte 1 de 2.)

```
27    }
28
29    // establecer horas trabajadas del empleado por horas
30    public void establecerHoras( double horasTrabajadas )
31    {
32       horas = ( horasTrabajadas >= 0.0 && horasTrabajadas <= 168.0 ) ?
33          horasTrabajadas : 0.0;
34    }
35
36    // devolver horas trabajadas
37    public double obtenerHoras()
38    {
39       return horas;
40    }
41
42    // calcular pago del empleado por horas;
43    // sobrescribir el método abstracto ingresos en Empleado
44    public double ingresos()
45    {
46       if ( horas <= 40 )  // sin tiempo extra
47          return sueldo * horas;
48       else
49          return 40 * sueldo + ( horas - 40 ) * sueldo * 1.5;
50    }
51
52    // devolver la representación String del objeto EmpleadoPorHoras
53    public String toString()
54    {
55       return "\nempleado por horas: " + super.toString();
56    }
57
58 } // fin de la clase EmpleadoPorHoras
```

Figura 10.14 La clase **EmpleadoPorHoras** que extiende a **Empleado**. (Parte 2 de 2.)

La clase **EmpleadoPorComision** (figura 10.15) extiende a la clase **Empleado** (línea 4). Esta clase incluye un constructor (líneas 9 a 16) que toma un primer nombre, un apellido paterno, un número de seguro social, una cantidad de ventas y una tasa de comisión como argumentos; métodos *establecer* (líneas 19 a 22 y 31 a 34) para asignar nuevos valores a las variables de instancia **tasaComision** y **ventasTotales**, respectivamente; métodos *obtener* (líneas 25 a 28 y 37 a 40) para devolver los valores de estas variables de instancia; el método **ingresos** (líneas 44 a 47) que calcula los ingresos de un **EmpleadoPorComision**; y el método **toString** (líneas 50 a 53), que devuelve el tipo del empleado, en este caso **"empleado por comisión: "** e información específica de ese empleado. El constructor de **EmpleadoPorComision** también pasa el primer nombre, apellido paterno y número de seguro social al constructor de **Empleado** para inicializar los campos heredados (línea 13). El método **toString** llama al método **toString** de la superclase (línea 52) para obtener la información específica del **Empleado** (es decir, primer nombre, apellido paterno y número de seguro social).

```
1  // Fig. 10.15: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision que extiende a Empleado.
3
4  public class EmpleadoPorComision extends Empleado {
5     private double ventasTotales;       // ventas totales por semana
6     private double tasaComision;        // porcentaje de comisión
7
8     // constructor
9     public EmpleadoPorComision( String nombre, String apellido,
10       String numeroSeguroSocial,
11       double ventasTotalesSemana, double porcentaje )
12    {
13       super( nombre, apellido, numeroSeguroSocial );
14       establecerVentasTotales( ventasTotalesSemana );
15       establecerTasaComision( porcentaje );
16    }
17
18    // establecer la tasa de comisión del empleado
19    public void establecerTasaComision( double tasa )
20    {
21       tasaComision = ( tasa > 0.0 && tasa < 1.0 ) ? tasa : 0.0;
22    }
23
24    // devolver la tasa de comisión del empleado
25    public double obtenerTasaComision()
26    {
27       return tasaComision;
28    }
29
30    // establecer salario base semanal del empleado por comisión
31    public void establecerVentasTotales( double ventas )
32    {
33       ventasTotales = ventas < 0.0 ? 0.0 : ventas;
34    }
35
36    // devolver cantidad de ventas totales del empleado por comisión
37    public double obtenerVentasTotales()
38    {
39       return ventasTotales;
40    }
41
42    // calcular pago del empleado por comisión;
43    // sobrescribir el método abstracto ingresos en Empleado
44    public double ingresos()
45    {
46       return obtenerTasaComision() * obtenerVentasTotales();
47    }
48
49    // devolver la representación String del objeto EmpleadoPorComision
50    public String toString()
51    {
52       return "\nempleado por comisión: " + super.toString();
53    }
54
55 } // fin de la clase EmpleadoPorComision
```

Figura 10.15 La clase **EmpleadoPorComision** que extiende a **Empleado**. (Parte 1 de 2.)

Figura 10.15 La clase **EmpleadoPorComision** que extiende a **Empleado**. (Parte 2 de 2.)

La clase **EmpleadoBaseMasComision** (figura 10.16) extiende a la clase **EmpleadoPorComision** (línea 4) y, por lo tanto, es también una subclase indirecta de la clase **Empleado**. La clase **EmpleadoBaseMasComision** tiene un constructor (líneas 8 a 14) que toma como argumentos un primer nombre, apellido

http://librosysolucionarios.net

paterno, número de seguro social, salario base, cantidad de ventas y tasa de comisión; un método *establecer* (líneas 17 a 20) para asignar un nuevo valor a la variable de instancia `salarioBase`; un método *obtener* (líneas 23 a 26) para devolver el valor de `salarioBase`; el método `ingresos` (líneas 30 a 33) que calcula los ingresos de un `EmpleadoBaseMasComision`; y el método `toString` (líneas 36 a 41), que devuelve el tipo del empleado, en este caso `"empleado con salario base más comisión: "` y utiliza los métodos de la clase `Empleado` para obtener el nombre del trabajador y su número de seguro social. La construcción de `EmpleadoBaseMasComision` pasa el primer nombre, apellido paterno, número de seguro social, cantidad de ventas y tasa de comisión al constructor de `EmpleadoPorComision` para inicializar los miembros heredados (línea 12). Observe que en la línea 32 del método `ingresos` se hace una llamada al método `ingresos` de la superclase `EmpleadoPorComision`, para calcular la porción de los ingresos correspondiente a la comisión. Éste es un excelente ejemplo de reutilización de código.

```
1  // Fig. 10.16: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision que extiende a EmpleadoPorComision.
3
4  public class EmpleadoBaseMasComision extends EmpleadoPorComision {
5     private double salarioBase;  // salario base por semana
6
7     // constructor
8     public EmpleadoBaseMasComision( String nombre, String apellido,
9        String numeroSeguroSocial, double cantidadVentasTotales,
10       double tasa, double cantidadSalarioBase )
11    {
12       super( nombre, apellido, numeroSeguroSocial, cantidadVentasTotales, tasa );
13       establecerSalarioBase( cantidadSalarioBase );
14    }
15
16    // establecer el salario base del empleado con salario base más comisión
17    public void establecerSalarioBase( double salario )
18    {
19       salarioBase = salario < 0.0 ? 0.0 : salario;
20    }
21
22    // devolver el salario base del empleado con salario base más comisión
23    public double obtenerSalarioBase()
24    {
25       return salarioBase;
26    }
27
28    // calcular los ingresos del empleado con salario base más comisión;
29    // sobrescribir el método ingresos en EmpleadoPorComision
30    public double ingresos()
31    {
32       return obtenerSalarioBase() + super.ingresos();
33    }
34
35    // devolver la representación String de EmpleadoBaseMasComision
36    public String toString()
37    {
38       return "\nempleado con salario base más comisión: " +
39          super.obtenerPrimerNombre() + " " + super.obtenerApellidoPaterno() +
40          "\nnúmero de seguro social: " + super.obtenerNumeroSeguroSocial();
41    }
42
43 } // fin de la clase EmpleadoBaseMasComision
```

Figura 10.16 La clase `EmpleadoBaseMasComision` que extiende a `EmpleadoPorComision`.

El programa de la figura 10.17 prueba nuestra jerarquía de `Empleado` e incrementa el salario base de cada `EmpleadoBaseMasComision` en un 10%. En la línea 13 se crea el arreglo `empleados` tipo `Empleado` de cuatro elementos, el cual almacena las referencias a objetos `Empleado`. En las líneas 16 a 22 se llena el arreglo con referencias a objetos de las clases `EmpleadoAsalariado`, `EmpleadoPorComision`, `EmpleadoBaseMasComision` y `EmpleadoPorHoras`. El ciclo `for` (líneas 28 a 50) itera a través del arreglo `empleados` y muestra el nombre de cada empleado, su número de seguro social y sus ingresos. En la línea 29 se invoca el método `toString` del objeto al que se refiera `empleados[ i ]`. La llamada a este método es otro ejemplo de vinculación dinámica.

En este ejemplo queremos realizar un procesamiento especial en los objetos `EmpleadoBaseMasComision`; al encontrarnos con estos objetos, queremos incrementar su salario base en un 10%. Como los empleados se procesan en forma polimórfica, no podemos estar seguros en cuanto al tipo de `Empleado` que se esté manipulando en un momento dado. Esto crea un problema, ya que se deben identificar los empleados `EmpleadoBaseMasComision` para poderles pagar correctamente. En la línea 32 se utiliza el operador ***`instanceof`*** para determinar si el tipo de cada objeto es compatible con el tipo `EmpleadoBaseMasComision`. La condición en la línea 32 es verdadera si el objeto referenciado por `empleados[ i ]` *es un* `EmpleadoBaseMasComision`. Esto también sería verdadero para cualquier objeto de una subclase de `EmpleadoBaseMasComision`. En las líneas 36 y 37 se realiza una conversión descendente de `empleados[ i ]`, del tipo `Empleado` al tipo `EmpleadoBaseMasComision`. Esta conversión se permite solamente si el objeto tiene una relación del tipo *es un* con `EmpleadoBaseMasComision`. La condición de la línea 32 asegura que éste sea el caso.

```
1  // Fig. 10.17: PruebaSistemaNomina.java
2  // Programa de prueba de la jerarquía Empleado.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaSistemaNomina {
7
8     public static void main( String[] args )
9     {
10       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
11
12       // crear arreglo tipo Empleado
13       Empleado empleados[] = new Empleado[ 4 ];
14
15       // inicializar arreglo con Empleados
16       empleados[ 0 ] = new EmpleadoAsalariado( "Juan", "Pérez",
17          "111-11-1111", 800.00 );
18       empleados[ 1 ] = new EmpleadoPorComision( "Arlae", "Razón",
19          "222-22-2222", 10000, .06 );
20       empleados[ 2 ] = new EmpleadoBaseMasComision( "Alfonso", "Romero",
21          "333-33-3333", 5000, .04, 300 );
22       empleados[ 3 ] = new EmpleadoPorHoras( "Isabel", "Romero",
23          "444-44-4444", 16.75, 40 );
24
25       String salida = "";
26
27       // procesar genéricamente cada elemento en el arreglo empleados
28       for ( int i = 0; i < empleados.length; i++ ) {
29          salida += empleados[ i ].toString();
30
31          // determinar si el elemento es un EmpleadoBaseMasComision
32          if ( empleados[ i ] instanceof EmpleadoBaseMasComision ) {
33
```

Figura 10.17 Programa de prueba para la jerarquía de la clase `Empleado`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
34              // conversión descendente de referencia a Empleado a
35              // referencia a EmpleadoBaseMasComision
36              EmpleadoBaseMasComision empleadoActual =
37                 ( EmpleadoBaseMasComision ) empleados[ i ];
38
39              double salarioBaseAnterior = empleadoActual.obtenerSalarioBase();
40              salida += "\nsalario base anterior: $" + salarioBaseAnterior;
41
42              empleadoActual.establecerSalarioBase( 1.10 * salarioBaseAnterior );
43              salida += "\nel nuevo salario base con aumento del 10% es: $" +
44                 empleadoActual.obtenerSalarioBase();
45
46           } // fin de instrucción if
47
48           salida += "\nganó $" + empleados[ i ].ingresos() + "\n";
49
50        } // fin de instrucción for
51
52        // obtener nombre del tipo de cada objeto en el arreglo empleados
53        for ( int j = 0; j < empleados.length; j++ )
54           salida += "\nEl empleado " + j + " es un " +
55              empleados[ j ].getClass().getName();
56
57        JOptionPane.showMessageDialog( null, salida );  // mostrar resultados
58        System.exit( 0 );
59
60     } // fin de main
61
62  } // fin de la clase PruebaSistemaNomina
```

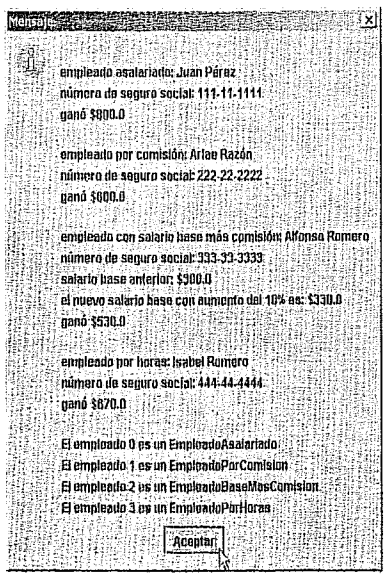


Figura 10.17 Programa de prueba para la jerarquía de la clase `Empleado`. (Parte 2 de 2)

http://librosysolucionarios.net

Error común de programación 10.5

En una operación de conversión descendente, si el tipo del objeto no tiene una relación es un con el tipo especificado en el operador de conversión, ocurre una excepción `ClassCastException`. *Un objeto puede convertirse sólo en su propio tipo o en el tipo de una de sus superclases.*

Si el valor devuelto por la instrucción `instanceof` en la línea 32 es `true`, la instrucción `if` (líneas 32 a 46) realiza el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. En las líneas 39 y 42 se invocan los métodos `obtenerSalarioBase` y `establecerSalarioBase` de `EmpleadoBaseMasComision` para recuperar y actualizar el salario del empleado. En la línea 48 se invoca el método `ingresos` en `empleado[ i ]`, el cual llama al método `ingresos` del objeto de la subclase correspondiente, utilizando vinculación dinámica.

El ciclo `for` (líneas 53 a 55) muestra el tipo de cada empleado como una cadena. Todo objeto en Java conoce su propia clase y puede acceder a esta información a través del método `getClass`, el cual heredan de la clase `Object` todas las clases. El método `getClass` devuelve un objeto de tipo `Class` (paquete `java.lang`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase. En la línea 55 se invoca el método `getClass` en el objeto para obtener la clase en tiempo de ejecución (objeto `Class`) de ese objeto. Luego se invoca el método `getName` en el objeto devuelto por `getClass`, para obtener el nombre de la clase. En la línea se muestra la cadena de salida, invocando al método `showMessageDialog` de la clase `JOptionPane`.

Observación de ingeniería de software 10.7

Java proporciona mecanismos para cargar clases en un programa de manera dinámica, para mejorar la funcionalidad de un programa en ejecución. En especial, el método estático `forName` *de la clase* `Class` *(paquete* `java.lang`*) puede utilizarse para cargar una clase y crear nuevos objetos de esa clase para utilizarlos en un programa. (Este concepto está más allá del alcance de este libro. Si desea más información, consulte la clase* `Class` *en la documentación en línea de la API.)*

10.8 Ejemplo práctico: Creación y uso de interfaces

En nuestro siguiente ejemplo (figuras 10.18 a 10.20) analizaremos, por última vez, la jerarquía `Punto/Circulo/Cilindro`, reemplazando la clase abstracta `Figura` con la interfaz `Figura` (figura 10.18). La declaración de una interfaz comienza con la palabra clave `interface` y contiene un conjunto de métodos `public abstract`. Las interfaces pueden contener también datos `public static final`. Para utilizar una interfaz, una clase debe especificar que implementará a esa interfaz (mediante `implements`) y debe declarar cada método en la interfaz con la firma especificada en la declaración de la interfaz. Si la clase no implementa a ninguno de los métodos de la interfaz, significa que la clase es abstracta y debe declararse como `abstract`. Implementar una interfaz es como firmar un contrato con el compilador que diga, "Declararé todos los métodos especificados por la interfaz".

Error común de programación 10.6

El no implementar uno de los métodos de una interfaz en una clase que implemente a esa interfaz resulta en un error de compilación, el cual indica que la clase debe declararse como `abstract`.

Empezamos a utilizar las interfaces cuando presentamos el manejo de eventos de la GUI en el capítulo 6, Métodos. Recuerde que nuestra clase de applet `Craps` de la figura 6.9 incluye la instrucción `implements ActionListener` (una interfaz en el paquete `java.awt.event`). Tuvimos que implementar el método `actionPerformed` en los applets con manejo de eventos para cumplir con el contrato de la interfaz `ActionListener`, el cual especifica que debe implementarse el método `actionPerformed`. Las interfaces son una parte importante del manejo de eventos de la GUI, como veremos en la siguiente sección.

Una interfaz se utiliza generalmente en lugar de una clase abstracta, cuando no hay una implementación predeterminada que heredar (es decir, no hay variables de instancia ni implementaciones de métodos predeterminados). Al igual que las clases `public abstract`, las interfaces son comúnmente tipos `public`, por lo que se declaran normalmente en archivos por sí solas, con el mismo nombre que la interfaz y la extensión de archivo `.java`.

La declaración de la interfaz `Figura` empieza, en la línea 4 de la figura 10.18. La interfaz `Figura` tiene tres métodos públicos abstractos `obtenerArea`, `obtenerVolumen` y `obtenerNombre`. Todos los

```
1  // Fig. 10.18: Figura.java
2  // Declaración de la interfaz Figura.
3
4  public interface Figura {
5     public double obtenerArea();     // calcular el área
6     public double obtenerVolumen();  // calcular el volumen
7     public String obtenerNombre();   // devolver el nombre de la figura
8
9  } // fin de la interfaz Figura
```

Figura 10.18 Declaración de la interfaz **Figura**.

métodos en una interfaz deben ser **public** y **abstract**, por lo que no necesitan declararse como tales. Por coincidencia, estos tres métodos no toman argumentos. Sin embargo, esto no es un requerimiento para los métodos en una interfaz. (De hecho, el método **actionPerformed** de la interfaz **ActionListener** requiere un argumento.)

En la línea 4 de la figura 10.19 se indica que la clase **Punto** extiende a la clase **Objeto** e implementa a la interfaz **Figura**. Java no permite a las subclases heredar de más de una superclase, pero sí permite que una clase herede de una superclase e implemente interfaces. La clase **Punto** implementa a los tres métodos en la interfaz. El método **obtenerArea** se declara en las líneas 47 a 50. El método **volumen** se declara en las líneas 53 a 56. El método **obtenerNombre** se declara en las líneas 59 a 62. Estos tres métodos satisfacen el requerimiento de implementación para los tres métodos en la interfaz. Por lo tanto, hemos cumplido nuestro contrato con el compilador.

```
1  // Fig. 10.19: Punto.java
2  // La declaración de la clase Punto que implementa a la interfaz Figura.
3
4  public class Punto extends Object implements Figura {
5     private int x;  // parte x del par de coordenadas
6     private int y;  // parte y del par de coordenadas
7
8     // constructor sin argumentos; el valor predeterminado de x e y es 0
9     public Punto()
10    {
11       // la llamada implícita al constructor de Object ocurre aquí
12    }
13
14    // constructor
15    public Punto( int valorX, int valorY )
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18       x = valorX;  // no hay necesidad de validación
19       y = valorY;  // no hay necesidad de validación
20    }
21
22    // establecer x en el par de coordenadas
23    public void establecerX( int valorX )
24    {
25       x = valorX;  // no hay necesidad de validación
26    }
27
28    // devolver x del par de coordenadas
29    public int obtenerX()
```

Figura 10.19 Implementación de **Punto** de la interfaz **Figura**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
30    {
31       return x;
32    }
33
34    // establecer y en el par de coordenadas
35    public void establecerY( int valorY )
36    {
37       y = valorY;  // no hay necesidad de validación
38    }
39
40    // devolver y del par de coordenadas
41    public int obtenerY()
42    {
43       return y;
44    }
45
46    // declarar el método abstracto obtenerArea
47    public double obtenerArea()
48    {
49       return 0.0;
50    }
51
52    // declarar el método abstracto obtenerVolumen
53    public double obtenerVolumen()
54    {
55       return 0.0;
56    }
57
58    // sobrescribir el método abstracto obtenerNombre para que devuelva "Punto"
59    public String obtenerNombre()
60    {
61       return "Punto";
62    }
63
64    // sobrescribir toString para que devuelva la representación String de Punto
65    public String toString()
66    {
67       return "[" + obtenerX() + ", " + obtenerY() + "]";
68    }
69
70 } // fin de la clase Punto
```

Figura 10.19 Implementación de **Punto** de la interfaz **Figura**. (Parte 2 de 2.)

Cuando una clase implementa a una interfaz se aplica la misma relación del tipo *es un* proporcionada por la herencia. Por ejemplo, la clase **Punto** implementa a **Figura**. Por lo tanto, un objeto **Punto** *es una* **Figura**. De hecho, los objetos de cualquier clase que extienda a **Punto** son también objetos **Figura**. Mediante el uso de esta relación hemos mantenido las declaraciones originales de la clase **Circulo** y la clase **Cilindro** de la sección 10.5. Para demostrar que puede utilizarse una interfaz en vez de una clase abstracta para procesar objetos **Figura** mediante el polimorfismo, la clase **PruebaInterfaz** (figura 10.20) utiliza un método **main** idéntico al de la clase **PruebaHerenciaAbstracta** (figura 10.10). Observe que la salida del programa en la figura 10.20 es idéntica a la de la figura 10.10. Además, observe que en la línea 38 se invoca el método **toString** a través de una referencia a la interfaz **Figura**, aún y cuando **toString** no está declarado en esta interfaz.

Observación de ingeniería de software 10.8

Todos los métodos de la clase `Object` *pueden llamarse mediante el uso de una referencia de un tipo de interfaz. Una referencia se refiere a un objeto, y todos los objetos heredan los métodos de la clase* `Object`.

```
1  // Fig. 10.20: PruebaInterfaz.java
2  // Prueba de la jerarquía Punto, Circulo, Cilindro con la interfaz Figura.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class PruebaInterfaz {
7
8     public static void main( String args[] )
9     {
10       // establecer formato de número de punto flotante
11       DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
12
13       // crear los objetos Punto, Circulo y Cilindro
14       Punto punto = new Punto( 7, 11 );
15       Circulo circulo = new Circulo( 22, 8, 3.5 );
16       Cilindro cilindro = new Cilindro( 20, 30, 3.3, 10.75 );
17
18       // obtener nombre y representación de cadena de cada objeto
19       String salida = punto.obtenerNombre() + ": " + punto + "\n" +
20          circulo.obtenerNombre() + ": " + circulo + "\n" +
21          cilindro.obtenerNombre() + ": " + cilindro + "\n";
22
23       Figura arregloDeFiguras[] = new Figura[ 3 ]; // crear arreglo tipo Figura
24
25       // apuntar arregloDeFiguras[ 0 ] al objeto de la subclase Punto
26       arregloDeFiguras[ 0 ] = punto;
27
28       // apuntar arregloDeFiguras[ 1 ] al objeto de la subclase Circulo
29       arregloDeFiguras[ 1 ] = circulo;
30
31       // apuntar arregloDeFiguras[ 2 ] al objeto de la subclase Cilindro
32       arregloDeFiguras[ 2 ] = cilindro;
33
34       // iterar a través de arregloDeFiguras para obtener nombre, representación
35       // de cadena, área y volumen para todos los objetos Figura en el arreglo
36       for ( int i = 0; i < arregloDeFiguras.length; i++ ) {
37          salida += "\n\n" + arregloDeFiguras[ i ].obtenerNombre() + ": " +
38             arregloDeFiguras[ i ].toString() + "\nArea = " +
39             dosDigitos.format( arregloDeFiguras[ i ].obtenerArea() ) +
40             "\nVolumen = " +
41             dosDigitos.format( arregloDeFiguras[ i ].obtenerVolumen() );
42       }
43
44       JOptionPane.showMessageDialog( null, salida );  // mostrar resultados
45
46       System.exit( 0 );
47
48    } // fin de main
49
50 } // fin de la clase PruebaInterfaz
```

Figura 10.20 La jerarquía `Punto/Circulo/Cilindro` con la interfaz `Figura`. (Parte 1 de 2.)

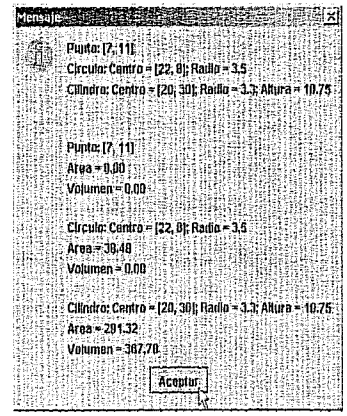


Figura 10.20 La jerarquía `Punto/Circulo/Cilindro` con la interfaz `Figura`. (Parte 2 de 2.)

Implementación de múltiples interfaces

Uno de los beneficios del uso de interfaces es que una clase puede implementar todas las interfaces que necesite, además de extender a una clase. Para implementar más de una interfaz, simplemente debe proporcionar una lista de nombres de interfaces separados por comas, después de la palabra clave `implements` en la declaración de la clase. Esto es particularmente útil en el mecanismo de manejo de eventos de la GUI. Una clase que implementa a más de una interfaz de escucha de eventos (como `ActionListener` en los ejemplos anteriores) puede procesar distintos tipos de eventos de la GUI, como veremos en los capítulos 13 y 14.

Declaración de constantes con interfaces

Otro uso de las interfaces es para declarar un conjunto de constantes que pueden utilizarse en muchas declaraciones de clases. Considere la interface `Constantes`:

```
public interface Constantes {
 public static final int UNO = 1;
 public static final int DOS = 2;
 public static final int TRES = 3;

}
```

Las clases que implementan a la interfaz `Constantes` pueden usar a `UNO`, `DOS` y `TRES` en cualquier parte de la declaración de la clase. Inclusive, una clase puede utilizar estas constantes si importa la interfaz y después hace referencia a cada constante como `Constantes.UNO`, `Constantes.DOS` y `Constantes.TRES`. Como no hay métodos declarados en esta interfaz, una clase que implemente a esta interfaz no tiene que implementar ningún método.

10.9 Clases anidadas

Todas las clases que hemos descrito hasta ahora son *clases de nivel superior* (es decir, no están declaradas dentro de una clase o un método). Por ejemplo, si un archivo contenía dos clases, una no estaba *anidada* en el cuerpo de la otra. Java permite el uso de *clases anidadas* (clases que se declaran dentro de otras clases). Las clases anidadas pueden ser `static`. Las clases de este tipo que no son anidadas se llaman *clases internas*. Estas clases se utilizan principalmente en el manejo de eventos. Sin embargo, tienen otros beneficios. Por ejemplo, la implementación del tipo de datos abstracto llamado cola, que se describió en la sección 8.6, podría utilizar una clase interna para representar a los objetos que almacenan cada elemento actualmente en la cola. Sólo la estructura de datos tipo cola requiere saber cómo se almacenan los objetos internamente, por lo que la implementación puede ocultarse declarando una clase interna como parte de la clase `Cola`.

http://librosysolucionarios.net

El siguiente ejemplo no sólo demuestra el uso de las declaraciones de clases anidadas con clases internas, sino que también demuestra el uso de una aplicación que se ejecuta en su propia ventana. Una vez que usted complete este ejemplo, podría utilizar en sus aplicaciones las técnicas de la GUI que se han mostrado, hasta ahora, sólo en los applets.

Recordemos el applet de las figuras 8.7 y 8.8, en el que creamos y utilizamos un objeto de la clase **Tiempo3** para mantener la hora que el usuario del applet podía manipular a través de la GUI proporcionada por el applet. Ahora implementaremos nuevamente el programa como una aplicación que se ejecuta en su propia ventana. Modificamos la clase **Tiempo3** y creamos una nueva clase llamada **Tiempo** (figura 10.21), con pocas diferencias. La clase **Tiempo** cambia el nombre de **aStringEstandar** por el de **toString**, de manera que el formato de hora estándar sea la representación de cadena predeterminada de un objeto **Tiempo**. Además, en la clase **Tiempo3** cada uno de los métodos **aStringUniversal** y **aStringEstandar** creaban un objeto **DecimalFormat** cada vez que eran llamados. La clase **Tiempo** declara una variable **DecimalFormat** en la línea 11 como un campo estático de la clase. Los métodos **aStringUniversal** y **toString** utilizan este objeto para el formato. (*Nota:* No hablaremos aquí sobre la clase **Tiempo**, ya que vimos todas sus características en el capítulo 8.)

```
1  // Fig. 10.21: Tiempo.java
2  // Declaración de la clase Tiempo con métodos establecer y obtener.
3  import java.text.DecimalFormat;
4
5  public class Tiempo {
6     private int hora;       // 0 - 23
7     private int minuto;     // 0 - 59
8     private int segundo;    // 0 - 59
9
10    // un objeto de formato para compartir en toString y aStringUniversal
11    private static DecimalFormat dosDigitos = new DecimalFormat( "00" );
12
13    // el constructor de Tiempo inicializa cada variable de instancia en cero;
14    // se asegura que cada objeto Tiempo inicie en un estado consistente
15    public Tiempo()
16    {
17       this( 0, 0, 0 ); // invocar al constructor de Tiempo con tres argumentos
18    }
19
20    // constructor de Tiempo: se proporciona hora; minuto y segundo con va-
         lor predeterminado de 0
21    public Tiempo( int h )
22    {
23       this( h, 0, 0 ); // invocar al constructor de Tiempo con tres argumentos
24    }
25
26    // constructor de Tiempo: se proporcionan hora y minuto, segundo con
         valor predeterminado de 0
27    public Tiempo( int h, int m )
28    {
29       this( h, m, 0 ); // invocar al constructor de Tiempo con tres argumentos
30    }
31
32    // constructor de Tiempo: se proporcionan hora, minuto y segundo
33    public Tiempo( int h, int m, int s )
34    {
35       establecerTiempo( h, m, s );
```

Figura 10.21 Declaración de la clase **Tiempo**. (Parte 1 de 3.)

http://librosysolucionarios.net


```
36    }
37
38    // constructor de Tiempo: se suministra otro objeto Tiempo3
39    public Tiempo( Tiempo tiempo )
40    {
41       // invocar al constructor de Tiempo con tres argumentos
42       this( tiempo.obtenerHora(), tiempo.obtenerMinuto(),
         tiempo.obtenerSegundo() );
43    }
44
45    // Métodos Establecer
46    // establecer un nuevo valor de tiempo, utilizando la hora universal; realizar
47    // comprobaciones de validez en los datos; establecer valores inválidos en cero
48    public void establecerTiempo( int h, int m, int s )
49    {
50       establecerHora( h );    // establecer la hora
51       establecerMinuto( m );  // establecer el minuto
52       establecerSegundo( s ); // establecer el segundo
53    }
54
55    // validar y establecer hora
56    public void establecerHora( int h )
57    {
58       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
59    }
60
61    // validar y establecer minuto
62    public void establecerMinuto( int m )
63    {
64       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
65    }
66
67    // validar y establecer segundo
68    public void establecerSegundo( int s )
69    {
70       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
71    }
72
73    // Métodos Obtener
74    // obtener valor de hora
75    public int obtenerHora()
76    {
77       return hora;
78    }
79
80    // obtener valor de minuto
81    public int obtenerMinuto()
82    {
83       return minuto;
84    }
85
86    // obtener valor de segundo
87    public int obtenerSegundo()
88    {
89       return segundo;
```

Figura 10.21 Declaración de la clase **Tiempo**. (Parte 2 de 3.)

```
90    }
91
92    // convertir a String en formato de hora universal
93    public String aStringUniversal()
94    {
95       return dosDigitos.format( obtenerHora() ) + ":" +
96          dosDigitos.format( obtenerMinuto() ) + ":" +
97          dosDigitos.format( obtenerSegundo() );
98    }
99
100   // convertir a String en formato de hora estándar
101   public String toString()
102   {
103      return ( ( obtenerHora() == 12 || obtenerHora() == 0 ) ?
104         12 : obtenerHora() % 12 ) + ":" + dosDigitos.format( obtenerMinuto() ) +
105         ":" + dosDigitos.format( obtenerSegundo() ) +
106         ( obtenerHora() < 12 ? " AM" : " PM" );
107   }
108
109 } // fin de la clase Tiempo
```

Figura 10.21 Declaración de la clase **Tiempo**. (Parte 3 de 3.)

En la figura 10.22 se declara la clase **VentanaPruebaTiempo** como una aplicación. Esta aplicación se ejecuta en su propia ventana y proporciona una GUI que permite al usuario cambiar la hora. En este ejemplo utilizamos una clase interna para implementar los manejadores de evento para los diversos componentes de la GUI de la ventana.

```
1  // Fig. 10.22: VentanaPruebaTiempo.java
2  // Declaraciones de clase interna utilizadas para crear manejadores de eventos.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class VentanaPruebaTiempo extends JFrame {
8     private Tiempo tiempo;
9     private JLabel horaEtiqueta, minutoEtiqueta, segundoEtiqueta;
10    private JTextField horaCampo, minutoCampo, segundoCampo, pantallaCampo;
11    private JButton salirBoton;
12
13    // configurar la GUI
14    public VentanaPruebaTiempo()
15    {
16       // llamar al constructor de JFrame para establecer cadena de barra de título
17       super( "Demostración de clase interna" );
18
19       tiempo = new Tiempo();  // crear objeto Tiempo
20
21       // usar método heredado getContentPane para obtener panel de
            contenido de la ventana
22       Container contenedor = getContentPane();
23       contenedor.setLayout( new FlowLayout() );  // cambiar esquema
24
```

Figura 10.22 Clase interna utilizada para el manejo de eventos en una aplicación de ventana. (Parte 1 de 4.)

http://librosysolucionarios.net


```
25       // preparar horaEtiqueta y horaCampo
26       horaEtiqueta = new JLabel( "Ajuste hora" );
27       horaCampo = new JTextField( 10 );
28       contenedor.add( horaEtiqueta );
29       contenedor.add( horaCampo );
30
31       // preparar minutoEtiqueta y minutoCampo
32       minutoEtiqueta = new JLabel( "Ajuste minuto" );
33       minutoCampo = new JTextField( 10 );
34       contenedor.add( minutoEtiqueta );
35       contenedor.add( minutoCampo );
36
37       // preparar segundoEtiqueta y segundoCampo
38       segundoEtiqueta = new JLabel( "Ajuste segundo" );
39       segundoCampo = new JTextField( 10 );
40       contenedor.add( segundoEtiqueta );
41       contenedor.add( segundoCampo );
42
43       // establecer pantallaCampo
44       pantallaCampo = new JTextField( 30 );
45       pantallaCampo.setEditable( false );
46       contenedor.add( pantallaCampo );
47
48       // establecer salirBoton
49       salirBoton = new JButton( "Salir" );
50       contenedor.add( salirBoton );
51
52       // crear una instancia de la clase interna ActionEventHandler
53       ActionEventHandler manejador = new ActionEventHandler();
54
55       // registrar manejadores de eventos; el objeto referenciado por manejador
56       // es el ActionListener, el cual contiene el método actionPerformed
57       // que será llamado para manejar los eventos de acción generados por
58       // horaCampo, minutoCampo, segundoCampo y salirBoton
59       horaCampo.addActionListener( manejador );
60       minutoCampo.addActionListener( manejador );
61       segundoCampo.addActionListener( manejador );
62       salirBoton.addActionListener( manejador );
63
64    } // fin del constructor
65
66    // mostrar tiempo en pantallaCampo
67    public void mostrarTiempo()
68    {
69       pantallaCampo.setText( "La hora es: " + tiempo );
70    }
71
72    // iniciar aplicación: crear, ajustar tamaño y mostrar VentanaPruebaTiempo;
73    // cuando main termina, el programa continúa ejecutándose ya que
74    // se muestra una ventana mediante las instrucciones en main
75    public static void main( String args[] )
76    {
77       VentanaPruebaTiempo ventana = new VentanaPruebaTiempo();
78
```

Figura 10.22 Clase interna utilizada para el manejo de eventos en una aplicación de ventana. (Parte 2 de 4.)

```
79          ventana.setSize( 400, 140 );
80          ventana.setVisible( true );
81
82       } // fin de main
83
84       // declaración de clase interna para manejar eventos JTextField y JButton
85       private class ActionEventHandler implements ActionListener {
86
87          // método para manejar eventos de acción
88          public void actionPerformed( ActionEvent evento )
89          {
90             // el usuario oprimió salirBoton
91             if ( evento.getSource() == salirBoton )
92                System.exit( 0 );   // terminar la aplicación
93
94             // el usuario oprimió tecla Intro en horaCampo
95             else if ( evento.getSource() == horaCampo ) {
96                tiempo.establecerHora( Integer.parseInt(
97                   evento.getActionCommand() ) );
98                horaCampo.setText( "" );
99             }
100
101            // el usuario oprimió tecla Intro en minutoCampo
102            else if ( evento.getSource() == minutoCampo ) {
103               tiempo.establecerMinuto( Integer.parseInt(
104                  evento.getActionCommand() ) );
105               minutoCampo.setText( "" );
106            }
107
108            // el usuario oprimió tecla Intro en segundoCampo
109            else if ( evento.getSource() == segundoCampo ) {
110               tiempo.establecerSegundo( Integer.parseInt(
111                  evento.getActionCommand() ) );
112               segundoCampo.setText( "" );
113            }
114
115            mostrarTiempo();  // llamar a un método de la clase externa
116
117         } // fin del método actionPerformed
118
119      } // fin de la clase interna ActionEventHandler
120
121   } // fin de la clase VentanaPruebaTiempo
```

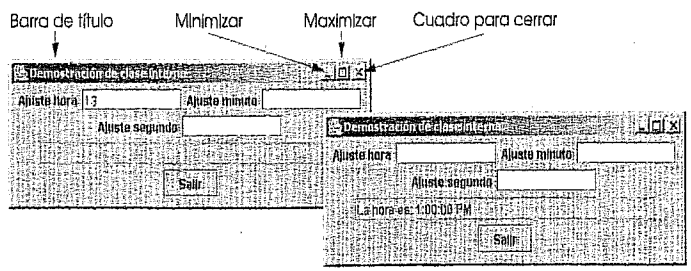


Figura 10.22 Clase interna utilizada para el manejo de eventos en una aplicación de ventana. (Parte 3 de 4.)

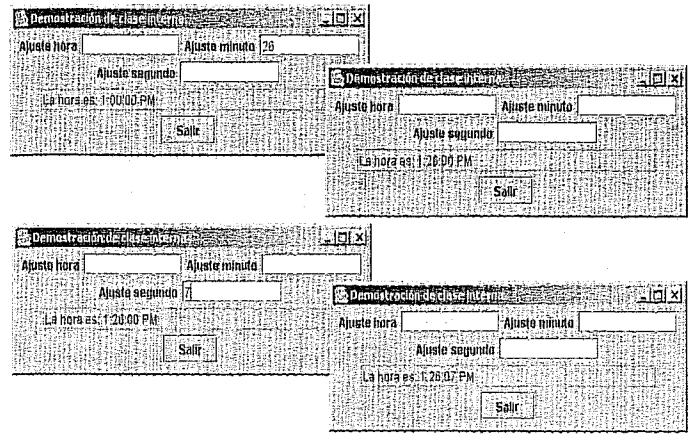


Figura 10.22 Clase interna utilizada para el manejo de eventos en una aplicación de ventana. (Parte 4 de 4.)

En la línea 7 de la figura 10.22 se indica que la clase `VentanaPruebaTiempo` extiende a la clase *`JFrame`* (del paquete `javax.swing`), en vez de extender a la clase `JApplet` (como se muestra en la figura 8.7). La superclase `JFrame` proporciona los atributos y comportamientos básicos de una ventana; una *barra de título* y botones para *minimizar*, *maximizar* y *cerrar* la ventana (como se muestra en la primera pantalla de la figura 10.22). La clase `VentanaPruebaTiempo` utiliza los mismos componentes de la GUI que el applet (subprograma) de la figura 8.7, excepto que el botón (línea 11) ahora se llama `salirBoton` y se utiliza para cerrar la aplicación.

El método `init` del applet se ha reemplazado por un constructor (líneas 11 a 64) que crea los componentes de la GUI de la ventana, a medida que la aplicación comienza a ejecutarse. El método `main` (líneas 75 a 82), que empieza la ejecución de la aplicación, crea un nuevo objeto de la clase `VentanaPruebaTiempo` que resulta en una llamada al constructor. Recuerde que `init` es un miembro especial, el cual se garantiza que será llamado cuando un applet empiece a ejecutarse. Sin embargo, este programa no es un applet, por lo que si declaramos el método `init`, éste no se llamará automáticamente.

En el constructor aparecen varias características nuevas. En la línea 17 se llama al constructor de la superclase `JFrame` con la cadena `"Demostración de clase interna"`. Esta cadena se muestra en la barra de título de la ventana mediante el constructor de la clase `JFrame`. En la línea 53 se crea un objeto de la clase interna `ActionEventHandler` (declarada dentro de la clase `VentanaPruebaTiempo` en las líneas 85 a 119) y se asigna su referencia a `manejador`. Esta referencia se pasa a cada una de las cuatro llamadas a `addActionListener` (líneas 59 a 62) que registran los manejadores de eventos para cada componente de la GUI que genera eventos en este ejemplo (`horaCampo`, `minutoCampo`, `segundoCampo` y `salirBoton`). Cada llamada a `addActionListener` requiere que se pase un objeto de tipo `ActionListener` como argumento. En realidad, `manejador` *es un* `ActionListener`. En la línea 85 (la primera línea de la declaración de la clase interna) se indica que la clase interna `ActionEventHandler` implementa a `ActionListener`. Por lo tanto, cada objeto de tipo `ActionEventHandler` *es un* `ActionListener`, y ¡se cumple con el requerimiento de que `addActionListener` se pase a un objeto de tipo `ActionListener`! La relación *es un* se utiliza en forma extensa en el mecanismo de manejo de eventos de la GUI, como veremos en los siguientes capítulos. La clase interna se declara como `private` ya que se utilizará solamente en esta declaración de clase. Las clases internas pueden declararse como `private`, `protected`, `public` o con acceso a nivel de paquete (el predeterminado).

Una clase interna tiene una relación especial con la clase externa que la declara. La clase interna puede tener acceso directo a todas las variables de instancia y los métodos de la clase externa. El método `action-`

http://librosysolucionarios.net

`Performed` (líneas 88 a 117) de la clase `ActionEventHandler` hace justo eso. En este método se utilizan las variables de instancia `tiempo`, `botonSalir`, `horaCampo`, `minutoCampo` y `segundoCampo`, al igual que el método `mostrarTiempo`. Observe que ninguno de estos métodos necesita una referencia al objeto de la clase externa.

Observación de ingeniería de software 10.9



Una clase interna puede tener acceso directo a las variables y métodos de su clase externa.

Observación de ingeniería de software 10.10



La referencia `this` que se utiliza en una clase interna se refiere al objeto actual de la clase interna que se está manipulando. Un método de la clase interna puede utilizar a la referencia `this` de su objeto de la clase externa si antepone a la palabra `this` el nombre de la clase externa y un punto, como en `NombreClaseExterna.this`.

Esta aplicación debe terminarse oprimiendo el botón `Salir`. Recuerde, una aplicación que muestra una ventana debe terminarse con una llamada a `System.exit(0)`, que es exactamente lo que hace el manejador de eventos para `salirBoton` (líneas 91 y 92). Además, observe que una ventana en Java tiene 0 píxeles de ancho, 0 píxeles de alto y no se muestra de manera predeterminada. En las líneas 79 y 80 se utilizan los métodos `setSize` y `setVisible` para cambiar el tamaño de la ventana y mostrarlo en la pantalla. Estos métodos se declaran en la clase `java.awt.Component`, heredada por la clase `JFrame` y heredada de nuevo por `VentanaPruebaTiempo`.

Clases internas anónimas

Una clase interna también puede declararse dentro del método de una clase. Dicha clase interna tiene acceso a los miembros de su clase externa. Sin embargo, tiene acceso limitado a las variables locales del método en el que está declarada.

Observación de ingeniería de software 10.11



Una clase interna declarada en un método puede acceder a las variables de instancia y los métodos del objeto de la clase externa que la declaró, así como a las variables locales `final` de ese método.

En la figura 10.23 modificamos la clase `VentanaPruebaTiempo` de la figura 10.22 para que utilice *clases internas anónimas*. Una clase interna anónima no tiene nombre, por lo que se crea un objeto de esta clase en el punto en que se declara en el programa. En el siguiente ejemplo demostraremos las clases internas anónimas de dos formas. Primero, utilizaremos *clases internas anónimas separadas que implementan a una interfaz* (`ActionListener`) para crear manejadores de eventos para cada objeto `JTextField`: `horaCampo`, `minutoCampo` y `segundoCampo`. También demostraremos cómo terminar una aplicación cuando el usuario haga clic en el cuadro para **cerrar** la ventana. En este caso, el manejador de eventos se declara como una *clase interna anónima que extiende a una clase* (`WindowAdapter`). Aquí utilizamos de nuevo a la clase `Tiempo` (figura 10.21). Además, quitamos el botón **Salir** de este ejemplo. El constructor de `VentanaPruebaTiempo2` invoca al método `crearGUI` (líneas 24 a 48) para crear la GUI; luego invoca al método `registrarManejadoresDeEventos` (líneas 51 a 103) para registrar el manejador de eventos para `horaCampo`, `minutoCampo` y `segundoCampo`.

```
1  // Fig. 10.23: VentanaPruebaTiempo2.java
2  // Demostración de los métodos establecer y obtener de la clase Tiempo
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class VentanaPruebaTiempo2 extends JFrame {
8     private Tiempo tiempo;
9     private JLabel horaEtiqueta, minutoEtiqueta, segundoEtiqueta;
10    private JTextField horaCampo, minutoCampo, segundoCampo, pantallaCampo;
```

Figura 10.23 Clases internas anónimas. (Parte 1 de 4.)

http://librosysolucionarios.net


```
11
12    // constructor
13    public VentanaPruebaTiempo2()
14    {
15       // llamar al constructor de JFrame para establecer cadena de barra de título
16       super( "Demostración de clase interna anónima" );
17
18       tiempo = new Tiempo();           // crear objeto Tiempo
19       crearGUI();                      // configurar la GUI
20       registrarManejadoresDeEventos(); // configurar el manejo de eventos
21    }
22
23    // crear componentes de la GUI y adjuntarlos al panel de contenido
24    private void crearGUI()
25    {
26       Container contenedor = getContentPane();
27       contenedor.setLayout( new FlowLayout() );
28
29       horaEtiqueta = new JLabel( "Ajuste hora" );
30       horaCampo = new JTextField( 10 );
31       contenedor.add( horaEtiqueta );
32       contenedor.add( horaCampo );
33
34       minutoEtiqueta = new JLabel( "Ajuste minuto" );
35       minutoCampo = new JTextField( 10 );
36       contenedor.add( minutoEtiqueta );
37       contenedor.add( minutoCampo );
38
39       segundoEtiqueta = new JLabel( "Ajuste segundo" );
40       segundoCampo = new JTextField( 10 );
41       contenedor.add( segundoEtiqueta );
42       contenedor.add( segundoCampo );
43
44       pantallaCampo = new JTextField( 30 );
45       pantallaCampo.setEditable( false );
46       contenedor.add( pantallaCampo );
47
48    } // fin del método crearGUI
49
50    // registrar manejadores de eventos para horaCampo, minutoCampo y segundoCampo
51    private void registrarManejadoresDeEventos()
52    {
53       // registrar manejador de eventos para horaCampo
54       horaCampo.addActionListener(
55
56          new ActionListener() { // clase interna anónima
57
58             public void actionPerformed( ActionEvent evento )
59             {
60                tiempo.establecerHora( Integer.parseInt(
61                   evento.getActionCommand() ) );
62                horaCampo.setText( "" );
63                mostrarTiempo();
64             }
65
```

Figura 10.23 Clases internas anónimas. (Parte 2 de 4.)

```
66          } // fin de la clase interna anónima
67
68       ); // fin de la llamada a addActionListener para horaCampo
69
70       // registrar manejador de eventos para minutoCampo
71       minutoCampo.addActionListener(
72
73          new ActionListener() {  // clase interna anónima
74
75             public void actionPerformed( ActionEvent evento )
76             {
77                tiempo.establecerMinuto( Integer.parseInt(
78                   evento.getActionCommand() ) );
79                minutoCampo.setText( "" );
80                mostrarTiempo();
81             }
82
83          } // fin de la clase interna anónima
84
85       ); // fin de la llamada a addActionListener para minutoCampo
86
87       segundoCampo.addActionListener(
88
89          new ActionListener() {  // clase interna anónima
90
91             public void actionPerformed( ActionEvent evento )
92             {
93                tiempo.establecerSegundo( Integer.parseInt(
94                   evento.getActionCommand() ) );
95                segundoCampo.setText( "" );
96                mostrarTiempo();
97             }
98
99          } // fin de la clase interna anónima
100
101      ); // fin de la llamada a addActionListener para segundoCampo
102
103   } // fin del método registrarManejadoresDeEventos
104
105   // mostrar tiempo en pantallaCampo
106   public void mostrarTiempo()
107   {
108      pantallaCampo.setText( "La hora es: " + tiempo );
109   }
110
111   // crear objeto VentanaPruebaTiempo2, registrarse para sus eventos de ventana
112   // y mostrarlo para empezar la ejecución de la aplicación
113   public static void main( String args[] )
114   {
115      VentanaPruebaTiempo2 ventana = new VentanaPruebaTiempo2();
116
117      // registrar componente de escucha para evento windowClosing
118      ventana.addWindowListener(
119
```

Figura 10.23 Clases internas anónimas. (Parte 3 de 4.)

http://librosysolucionarios.net


```
120          // clase interna anónima para evento windowClosing
121          new WindowAdapter() {
122
123             // terminar la aplicación cuando el usuario cierra la ventana
124             public void windowClosing( WindowEvent evento )
125             {
126                System.exit( 0 );
127             }
128
129          } // fin de la clase interna anónima
130
131       ); // fin de la llamada a addWindowListener para ventana
132
133       ventana.setSize( 400, 105 );
134       ventana.setVisible( true );
135
136    } // fin de main
137
138 } // fin de la clase VentanaPruebaTiempo2
```

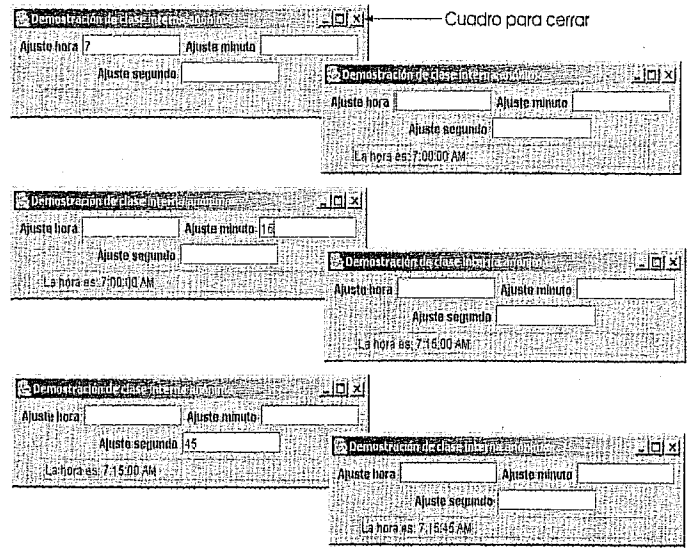


Figura 10.23 Clases internas anónimas. (Parte 4 de 4.)

Cada objeto `JTextField` que genera eventos en este programa cuenta con una clase interna anónima similar para manejar sus eventos, por lo que aquí hablaremos solamente de la clase interna anónima para `horaCampo`. En las líneas 54 a 68 se muestra una llamada al método `addActionListener` de `horaCampo`. El argumento para este método debe ser un objeto que *sea un* `ActionListener` (es decir, cualquier objeto de una clase que implemente a `ActionListener`). En las líneas 56 a 66 se muestra una expresión de creación de una instancia de clase, que declara a una clase interna anónima y crea un objeto de esa clase que se pasa como argumento para `addActionListener`. La sintaxis `ActionListener()` después de `new` comienza

la declaración de una clase interna anónima que implementa a la interfaz `ActionListener`. Esto es similar a empezar la declaración de una clase con:

```
public class MiManejador implements ActionListener {
```

Los paréntesis después de `ActionListener` indican una llamada al constructor predeterminado de la clase interna anónima.

La llave izquierda de apertura ({) en la línea 56 y la llave derecha de cierre (}) en la línea 66 delimitan el cuerpo de la clase. En las líneas 58 a 64 se declara el método `actionPerformed`, el cual es obligatorio en cualquier clase que implemente a `ActionListener`. El método `actionPerformed` se llama cuando el usuario oprime *Intro* mientras escribe en `horaCampo`. Observe que no se requiere de la lógica de la instrucción `if...else` en `actionPerformed` para determinar cuál componente de la GUI fue el origen del evento. Este manejador de eventos es específico para `horaCampo` y se llamará solamente en respuesta a un evento de acción en el objeto `horaCampo`. En las líneas 71 a 85 y 87 a 101 se declaran y registran manejadores de eventos separados para `minutoCampo` y `segundoCampo`, respectivamente.[1]

Observación de ingeniería de software 10.12



Cuando una clase interna anónima implementa a una interfaz, la clase debe implementar a todos los métodos en la interfaz.

El método `main` (líneas 113 a 136) crea una instancia de la clase `VentanaPruebaTiempo` (línea 115), ajusta el tamaño de la ventana (línea 133) y la muestra en pantalla (línea 134).

Las ventanas generan una variedad de eventos que veremos en el capítulo 14. Para este ejemplo, hablaremos sobre el único evento (un *evento de cierre de ventana*) que se genera cuando el usuario hace clic en el cuadro de cierre de la ventana. En las líneas 118 a 131 se permite al usuario terminar la aplicación haciendo clic en el cuadro para cerrar la ventana (que se muestra en la primera captura de pantalla). El método ***addWindowListener*** (heredado de `JFrame`) registra a un componente de escucha de eventos de ventana. El argumento para `addWindowListener` debe ser una referencia a un objeto que *sea un* ***WindowListener*** (una interfaz en el paquete `java.awt.event`); es decir, cualquier objeto de una clase que implemente a `WindowListener`. Sin embargo, hay siete métodos distintos que deben declararse en toda clase que implemente a `WindowListener`, y sólo necesitamos uno de ellos en este ejemplo: ***windowClosing***. Para las interfaces de manejo de eventos con más de un método, Java proporciona una clase correspondiente (conocida como *clase adaptadora*) que ya implementa a todos los métodos en la interfaz por usted, y proporciona un cuerpo vacío para cada método. Todo lo que usted necesita hacer es extender la clase adaptadora y sobrescribir sólo los métodos que necesite en su programa.

Error común de programación 10.7



Extender una clase adaptadora y escribir mal el nombre del método que usted va a sobrescribir es un error lógico. Su método simplemente se convierte en otro método más en la clase, que no será llamado para manejar eventos.

En las líneas 121 a 129 se declara una clase interna anónima y se crea un objeto de esa clase, el cual se pasa como argumento a `addWindowListener`. Después de `new`, la sintaxis `WindowAdapter()` comienza la declaración de una clase interna anónima que *extiende* a la clase `WindowAdapter`. Esto es similar a empezar la declaración de una clase con:

```
public class MiManejador extends WindowAdapter {
```

El compilador sabe cómo extender (en vez de implementar) a `WindowAdapter`, ya que es una clase. Los paréntesis después de `WindowAdapter` indican una llamada al constructor predeterminado de la clase interna anónima. La clase `WindowAdapter` implementa a la interfaz `WindowListener`, por lo que todo objeto `WindowAdapter` *es un* `WindowListener` (el tipo exacto requerido como argumento para `addWindowListener`). Desde luego que cualquier objeto de una clase que extienda a `WindowAdapter` *es un* `WindowAdapter`, lo cual implica que el objeto también *es un* `WindowListener`.

La llave izquierda de apertura ({) en la línea 121 y la llave derecha de cierre (}) en la línea 129 delimitan el cuerpo de la clase. En las líneas 124 a 127 se sobrescribe el método `windowClosing` de `WindowAdapter`, el cual se llama cuando el usuario hace clic en el cuadro para cerrar la ventana. En este ejemplo, `windowClosing` termina la aplicación. Debido a que el manejo de eventos que se muestra aquí es común para las ventanas de aplicación, la clase `JFrame` proporciona una técnica de método abreviado para registrar el manejador de eventos para un evento de cierre de ventana. La instrucción

```
ventana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

realiza las mismas tareas que las líneas 118 a 131. En este caso, especificamos como argumento la constante ***JFrame.EXIT_ON_CLOSE*** para indicar que el programa debe terminar cuando el usuario haga clic en el cuadro para cerrar. Otras opciones son `DO_NOTHING_ON_CLOSE` (ignorar el evento de cierre de ventana), `HIDE_ON_CLOSE` (ocultar la ventana, de manera que pueda volver a mostrarse posteriormente) y `DISPOSE_ON_CLOSE` (desechar la ventana, de manera que no pueda volver a mostrarse posteriormente). A partir de este punto, implementaremos nuestro propio `WindowListener` solamente si el programa debe realizar tareas adicionales cuando el usuario haga clic en el cuadro para cerrar la ventana. De no ser así, utilizaremos el método `setDefaultCloseOperation` para especificar que el programa debe terminar cuando el usuario haga clic en el cuadro para cerrar.

En los últimos dos ejemplos, hemos visto que las clases internas pueden utilizarse para crear manejadores de eventos, y que pueden declararse clases internas anónimas separadas para manejar eventos en forma individual para cada componente de la GUI. En los capítulos 13 y 14 volveremos a hablar sobre este concepto, al describir detalladamente el mecanismo para manejar eventos.

Observaciones acerca de las clases anidadas

Hay varios puntos de interés para los programadores, en relación con la declaración y el uso de las clases anidadas:

1. Al compilar una clase que contenga clases anidadas, se produce un archivo `.class` separado por cada clase. Las clases anidadas con nombres tienen un nombre de archivo de la forma *NombreClaseExterna*`$`*NombreClaseInterna*`.class`. Las clases internas anónimas tienen un nombre de archivo de la forma *NombreClaseExterna*`$#.class`, en donde `#` empieza en `1` y se incrementa por cada clase interna anónima que se encuentre durante la compilación.
2. Las clases internas con nombres pueden declararse como `public`, `protected`, con acceso a nivel de paquete o `private`, y están sujetas a las mismas restricciones de uso que tienen los demás miembros de las clases.
3. Una clase interna puede acceder a la referencia `this` de su clase externa con una expresión de la forma *NombreClaseExterna*`.this`.
4. La clase externa es responsable de crear los objetos de sus clases internas. Para crear un objeto de la clase interna de otra clase, primero se debe crear un objeto de la clase externa y asignar su referencia a una variable del tipo de la clase externa (a la que llamaremos `ref`). Después se debe utilizar una instrucción de la siguiente forma, para crear un objeto de una clase interna:

   NombreClaseExterior`.`*NombreClaseInterior* `refInterna = ref.new` *NombreClaseInterna*`();`

5. Una clase anidada `static` no requiere de la creación de un objeto de su clase externa (mientras que una clase interna sí). Una clase anidada `static` no tiene acceso a los miembros no `static` de su clase externa.

10.10 Clases de tipo de envoltura para los tipos primitivos

Como vimos en la sección 3.5, cada uno de los tipos primitivos tiene una *clase de envoltura de tipo* correspondiente (en el paquete `java.lang`). Estas clases son `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` y `Boolean`. Cada una de las clases de envoltura de tipos le permite manipular los valores de tipo primitivo como si fueran objetos. Por lo tanto, los valores de los tipos primitivos pueden procesarse mediante el

1. La sintaxis aquí mostrada para crear manejadores de eventos es similar al código que se genera mediante un IDE (entorno integrado de desarrollo) para Java. Generalmente, un IDE permite al programador diseñar una GUI en forma visual; después el IDE genera código para implementar la GUI. El programador simplemente inserta instrucciones en los métodos para manejo de eventos que declaran cómo manejar cada evento.

http://librosysolucionarios.net

polimorfismo si se mantienen como objetos de las clases de envoltura de tipos. Muchas de las clases que desarrollaremos o reutilizaremos manipulan y comparten objetos de la clase `Object`. Estas clases no pueden manipular variables de tipos primitivos mediante el polimorfismo, pero pueden manipular polimórficamente objetos de las clases de envolturas ya que cada clase se deriva en última instancia de la clase `Object`.

Cada una de las clases de envoltura de tipos numéricos (`Byte`, `Short`, `Integer`, `Long`, `Float` y `Double`) extiende a la clase `Number`. Cada una de las envolturas de tipos se declara como `final`, de manera que sus métodos sean implícitamente `final` y no puedan sobrescribirse. Observe que muchos de los métodos que procesan tipos primitivos se declaran como métodos `static` de las clases de envoltura de tipos. Si necesita manipular un valor primitivo en su programa, primero debe consultar la documentación de las clases de envoltura de tipos; tal vez el método que necesita ya esté declarado. Utilizaremos las clases de envoltura de tipos en forma polimórfica, en nuestro estudio de estructuras de datos en los capítulos 20 a 22.

10.11 (Ejemplo práctico opcional) Acerca de los objetos: Cómo incorporar la herencia a la simulación del elevador

Ahora regresaremos con nuestro diseño del simulador de elevador, para ver cómo podría éste beneficiarse de la herencia. Para aplicar la herencia, primero debemos buscar las características comunes entre las clases de la simulación. Comenzaremos por examinar las similitudes entre las clases `BotonPiso` y `BotonElevador`. En la figura 10.24 se muestran los atributos y operaciones de cada clase. Ambas clases tienen su atributo (`oprimido`) y operaciones (`oprimirBoton` y `restablecerBoton`) en común.

Podríamos vernos tentados a utilizar la herencia en esta situación (es decir, para extraer las características comunes entre las clases `BotonPiso` y `BotonElevador`, colocar estas características comunes en una superclase común llamada `Boton` y luego derivar las subclases `BotonPiso` y `BotonElevador` de la clase `Boton`). No obstante, si los objetos `BotonPiso` y `BotonElevador` tienen exactamente el mismo comportamiento, entonces no podemos justificar el uso de la herencia. De hecho, ¡no podemos ni siquiera justificar el uso de dos clases separadas para estos objetos! El `BotonPiso` indica al `Elevador` que se mueva al `Piso` opuesto. Como se muestra en el diagrama de actividad de la figura 5.29, el `BotonPiso` y el `BotonElevador` indican al `Elevador` que se mueva a un `Piso`. El `Elevador` se mueve en respuesta a la señal de un `BotonPiso` solamente si el `Elevador` se encuentra en el `Piso` opuesto al piso en que se hizo la solicitud y si el `Elevador` está inactivo. Éste se mueve en respuesta a una señal proveniente del `BotonElevador` sólo si el `Elevador` está inactivo. Sin embargo, ni el `BotonPiso` ni el `BotonElevador` le ordenan al `Elevador` que se mueva hacia el otro `Piso`. En vez de ello, el `Elevador` responde a la señal de un botón dependiendo de su estado actual. Cada botón tiene solamente un comportamiento: indicar al `Elevador` que se mueva. Los objetos `BotonPiso` y `BotonElevador` son realmente sólo objetos distintos de la misma clase, por lo que podemos combinar las clases `BotonPiso` y `BotonElevador` en la clase `Boton` y desechar las clases `BotonPiso` y `BotonElevador` de nuestro ejemplo práctico.

Observación de ingeniería de software 10.13



Si varios objetos tienen los mismos atributos y exhiben comportamientos idénticos, probablemente sean objetos de la misma clase. Antes de usar la herencia en sus programas, asegúrese de que cada clase tenga distintos atributos y/o exhiba distintos comportamientos, pero que siga "siendo un" objeto de su superclase.

Ahora empezaremos a buscar clases que exhiban comportamientos similares (pero no idénticos). En la sección 4.14 nos encontramos con el problema de representar la ubicación de la `Persona` (¿en qué `Piso` se

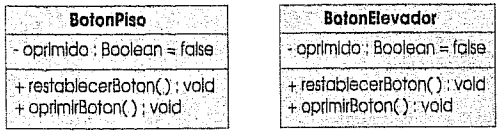


Figura 10.24 Atributos y operaciones de las clases `BotonPiso` y `BotonElevador`.

http://librosysolucionarios.net

encuentra la `Persona` cuando viaja en el `Elevador`?). Mediante el uso de la herencia, podemos modelar ahora una solución. Tanto el `Elevador` como los dos `Pisos` son ubicaciones en las que la `Persona` existe en el simulador. En otras palabras, el `Elevador` y los `Pisos` son *tipos de* ubicaciones, pero un `Elevador` ciertamente no es un `Piso`.

Modificaremos las clases `Elevador` y `Piso` para que hereden de una nueva superclase llamada `Ubicacion`. UML especifica una relación conocida como *generalización* para modelar la herencia. En la figura 10.25 se muestra el diagrama de clases que modela la generalización de la superclase `Ubicacion` y las subclases `Elevador` y `Piso`. Las flechas con punta sin relleno indican que las clases `Elevador` y `Piso` heredan de la clase `Ubicacion`. Esta clase contiene el atributo `private` llamado `nombreUbicacion`, el cual puede contener el valor `String` `"primerPiso"`, `"segundoPiso"` o `"elevador"`. Incluimos el método `establecerNombreUbicacion` para que cada clase pueda establecer el valor `String` apropiado. Observe que el método `establecerNombreUbicacion` tiene un modificador de acceso que no hemos visto aún: el signo de libras (#), que indica que el método `establecerNombreUbicacion` es `protected`, por lo que sólo las subclases `Elevador` y `Piso` pueden utilizar este método para establecer el valor de sus variables `nombreUbicacion`. Además, incluimos el método `public` llamado `obtenerNombreUbicacion`, para que cualquier objeto pueda obtener el nombre de la `Ubicacion`.

Buscaremos más similitudes entre las clases `Piso` y `Elevador`. De acuerdo con el diagrama de clases de la figura 6.20, ambas clases comparten el atributo entero `capacidad` (que es igual a `1`). Ahora, la clase `Ubicacion` contiene el atributo `private` llamado `capacidad`, el cual representa el máximo número de `Personas` que pueden ocupar esa ubicación. En nuestra simulación, este atributo no cambia en tiempo de ejecución, por lo que la clase `Ubicacion` declara al atributo `capacidad` como una constante mediante el uso del término `{frozen}` enseguida del atributo `capacidad`. La clase `Ubicacion` también contiene el método `public` `obtenerCapacidad`, el cual devuelve el valor de `capacidad`. La clase `Ubicacion` no requiere el método `establecerCapacidad`, ya que el atributo `capacidad` no puede cambiarse. Las subclases `Piso` y `Elevador` heredan el atributo `capacidad` y el método `obtenerCapacidad`.

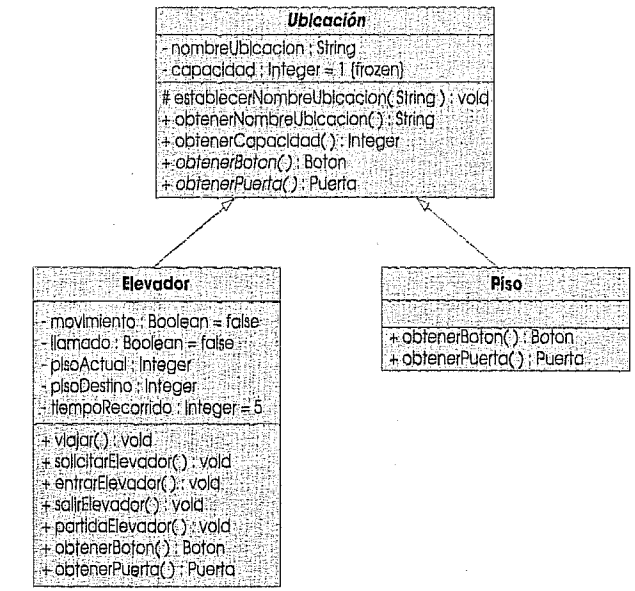


Figura 10.25 Diagrama de clases que modela la generalización de la superclase `Ubicacion` y las subclases `Elevador` y `Piso`.

Continuemos con nuestra búsqueda de similitudes. De acuerdo con la figura 3.19, la clase **Elevador** contiene referencias a su **Boton** y su **Puerta**. La clase **Piso** contiene referencias a su **Boton** y su **Puerta** mediante la asociación de **Piso** con la clase **ConductoElevador**; la clase **ModeloElevador** reúne a las clases **Piso** y **ConductoElevador**, y puede pasar una referencia **ConductoElevador** al constructor de la clase **Piso**. Mediante el uso de esta asociación, la clase **Piso** puede referenciar el **Boton** y la **Puerta** de **ConductoElevador**. Por lo tanto, en nuestra simulación la clase **Ubicacion** contendrá los métodos **public obtenerBoton** y **obtenerPuerta**, los cuales devuelven referencias **Boton** y **Puerta**, respectivamente. La clase **Piso** sobrescribe a estos métodos para devolver las referencias **Boton** y **Puerta** de ese **Piso**, y la clase **Elevador** sobrescribe estos métodos para devolver las referencias **Boton** y **Puerta** del **Elevador**.[2] Declaramos la clase **Ubicacion** como **abstract** para requerir que las subclases **Piso** y **Elevador** implementen los métodos **obtenerBoton** y **obtenerPuerta**. UML recomienda que coloquemos los nombres de las clases abstractas (y los métodos abstractos) en cursiva, por lo que en la figura 10.25 colocamos a la clase **Ubicacion** y sus métodos **obtenerBoton** y **obtenerPuerta** en cursiva. Sin embargo, los métodos **obtenerBoton** y **obtenerPuerta** no están en cursiva en las subclases **Piso** y **Elevador** (estos métodos son concretos ya que sobrescriben e implementan a los métodos abstractos). Cada método concreto tiene una implementación distinta aquí (es decir, la clase **Elevador** implementa a los métodos **obtenerBoton** y **obtenerPuerta** de manera distinta a la clase **Piso**). Observe que las clases **Elevador** y **Piso** proporcionan las operaciones **obtenerBoton** y **obtenerPiso** en su tercer compartimiento, ya que cada clase tiene distintas implementaciones del método sobrescrito.

En esta simulación, el uso de la herencia parece apropiado para diseñar a **Elevador** y **Piso**. Cada clase se representa a una **Ubicación** que la **Persona** puede ocupar. Sin embargo, la clase **Elevador** contiene atributos y métodos adicionales que la distinguen de la clase **Piso**.

Introduciremos una asociación entre la clase **Persona** y la clase **Ubicacion** que representa si la **Persona** se encuentra en el primer o segundo **Piso**, o dentro del **Elevador**.

Ahora concentrémonos en las clases **PuertaElevador** y **PuertaPiso**. Una vez más consideraremos si debemos usar la herencia con estas clases. En la figura 10.26 se muestra que los atributos y operaciones de cada clase son idénticos; ambas clases poseen el atributo **abierta** y las operaciones **abrirPuerta** y **cerrarPuerta**.

Sin embargo, con base en el enunciado del problema sabemos que **PuertaElevador** y **PuertaPiso** tienen distinto comportamiento. Cuando la **PuertaElevador** se abre, debe abrir a su vez la **PuertaPiso** para asegurarse que las puertas se abran en conjunto. La clase **PuertaPiso** tiene los atributos y operaciones básicas necesarias para todas las puertas, por lo que cambiaremos el nombre de **PuertaPiso** por el de **Puerta**. Esta nueva clase **Puerta** representa una generalización de las puertas en nuestro ejemplo práctico. La clase **PuertaElevador** representa un "caso especial" de la clase **Puerta** más general; al abrirse, una **PuertaElevador** tiene la responsabilidad de abrir otra **Puerta**. Por lo tanto, establecemos una relación de generalización entre las clases **PuertaElevador** y **Puerta**. La clase **PuertaElevador** sobrescribe a los métodos **abrirPuerta** y **cerrarPuerta** para implementar el comportamiento especializado (figura 10.27).

En la figura 10.28 se muestra un diagrama de clases actualizado de nuestro modelo, en el cual se reflejan estos cambios, se incorpora la herencia, se eliminan las clases **BotonPiso**, **BotonElevador** y **PuertaPiso**, y se introducen las clases **Boton**, **Puerta** y **Ubicacion**. Eliminamos del diagrama de clases la asociación entre **Persona** y **Elevador**, y la asociación entre **Persona** y **Piso**, ya que la referencia de **Ubicacion** de la **Persona** puede actuar ya sea como una referencia de **Elevador** o una referencia de **Piso**. Una **Persona** establece su referencia de **Ubicacion** a un **Piso** cuando la **Persona** camina en ese **Piso** particular. Por último, asignamos a la clase **Elevador** dos referencias de **Ubicacion**, las cuales representan el **Piso** actual del **Elevador** y el **Piso** de destino. (Originalmente utilizamos enteros para describir estas referencias en la figura 4.18.)

2. La mayoría de los métodos introducidos en este capítulo, como el método **obtenerRadio** (figura 10.8) o el método **obtenerAltura** (figura 10,9), devuelven valores primitivos, mientras que los métodos **obtenerBoton** y **obtenerPuerta** de las clases **Elevador** y **Piso** devuelven referencias a un objeto **Boton** y a un objeto **Piso**, respectivamente.

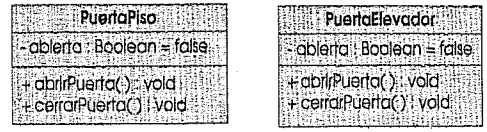


Figura 10.26 Atributos y operaciones de las clases **PuertaPiso** y **PuertaElevador**.

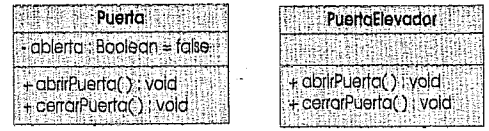


Figura 10.27 Generalización de la superclase **Puerta** y la subclase **PuertaElevador**.

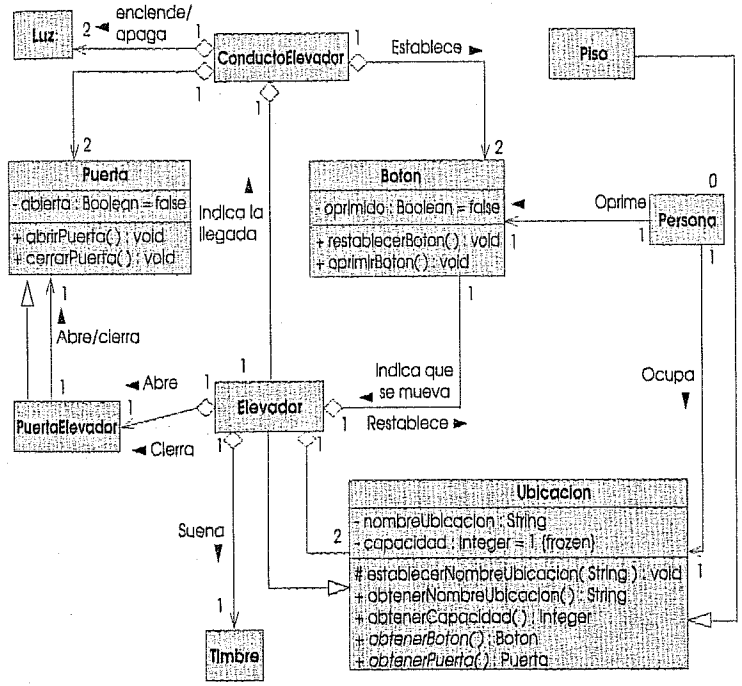


Figura 10.28 Diagrama de clases de nuestro simulador (incorporando la herencia).

Permitimos a una **Persona**, que ocupa una **Ubicacion**, que interactúe con varios de los objetos (accesibles desde esa **Ubicacion**) en el simulador. Por ejemplo, una **Persona** puede oprimir un **Boton** desde la **Ubicacion** específica de esa **Persona**. Una **Persona** puede ocupar sólo una **Ubicacion** a la vez, por lo que una **Persona** debe poder interactuar solamente con los objetos conocidos para esa **Ubicacion**.

http://librosysolucionarios.net

Utilizando su referencia de `Ubicacion`, una `Persona` no puede realizar una acción ilegal, como oprimir el `Boton` del primer `Piso` mientras se viaja en el `Elevador`. Esta restricción imita una situación real en la que una persona no puede oprimir un botón en ese piso mientras viaja en el elevador, y no puede oprimir un botón dentro del elevador mientras camina por un piso.

En el diagrama de clases de la figura 8.19 presentamos los atributos y las operaciones de las clases con modificadores de acceso. Ahora, en la figura 10.29 presentamos un diagrama de clases modificado en el que se incorpora la herencia. Este diagrama abreviado no muestra las relaciones de herencia, sino que muestra los atributos y métodos después de haber empleado la herencia en nuestro sistema. Este diagrama no incluye aquellos atributos que se muestran mediante las agregaciones en la figura 10.28, para las clases `ConductoElevador` y `Elevador`. Al igual que en la figura 4.18, optamos por ahorrar espacio en la figura 10.29 al no mostrar estos atributos adicionales; sin embargo, los incluiremos en la implementación en Java que viene en los apéndices.



Observación de ingeniería de software 10.14

Un diagrama de clases completo muestra todas las asociaciones entre clases, junto con todos los atributos y operaciones para cada clase. Cuando el número de atributos, métodos y asociaciones de las clases es substancial (como en las figuras 10.28 y 10.29), la práctica común es dividir esta información entre dos diagramas de clases: uno que se enfoque en las asociaciones y el otro que se enfoque en los atributos y métodos. Al crear dos diagramas de clases en esta manera se fomenta la legibilidad.

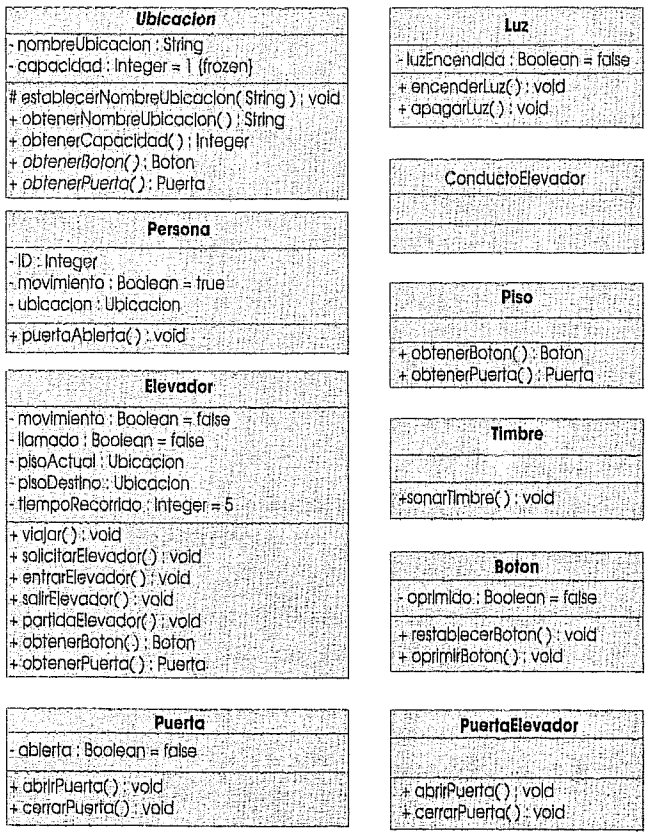


Figura 10.29 Diagrama de clases con atributos y operaciones (incorporando la herencia).

http://librosysolucionarios.net

La clase `Persona` ahora contiene una referencia a un objeto `Ubicación`, llamado `ubicacion`, que indica si la `Persona` está en un `Piso` o en el `Elevador`.

Implementación: ingeniería hacia delante (incorporando la herencia)

En la sección 8.17 utilizamos UML para expresar la estructura de clases de Java para nuestro simulador. Continuaremos con nuestra implementación incorporando la herencia, utilizando la clase `Elevador` como ejemplo.

1. Si una clase `B` es una generalización de la clase `A`, entonces la clase `A` extiende a la clase `B` en la declaración de la clase. Por ejemplo, la superclases abstracta `Ubicación` es una generalización de la clase `Elevador`, por lo que la declaración de clase debería ser:

```
public class Elevador extends Ubicacion {

   // constructor
   public Elevador() {}

}
```

2. Si la clase `B` es una clase abstracta, y la clase `A` es una subclase de la clase `B`, entonces la clase `A` debe implementar a los métodos abstractos de la clase `B` (si la clase `A` va a ser una clase concreta). Por ejemplo, la clase `Ubicacion` contiene los métodos abstractos `obtenerBoton` y `obtenerPuerta`, por lo que la clase `Elevador` debe implementar estos métodos ya que queremos instanciar un objeto `Elevador`. En la figura 10.30 se muestra el código en Java para la clase `Elevador` de las figuras 10.28 y 10.29. Observe que el método `obtenerBoton` (líneas 26 a 29) devuelve una referencia al objeto `Boton` del `Elevador`, y que el método `obtenerPuerta` (líneas 32 a 35) devuelve una referencia al objeto `Puerta` del `Elevador`; `Elevador` tiene asociaciones con ambos objetos, de acuerdo con el diagrama de clases de la figura 10.28. La clase `Elevador` hereda los campos `capacidad` y `nombreUbicacion`, y los métodos no abstractos `obtenerCapacidad`, `establecerNombreUbicacion` y `obtenerNombreUbicacion` de la superclase `Ubicacion`, por lo que no necesitamos declarar estos campos y métodos en la clase `Elevador`. La figura 10.29 especifica los atributos `movimiento`, `llamado`, `pisoActual`, `pisoDestino` y `tiempoRecorrido`, junto con las operaciones `viajar`, `solicitarElevador`, `entrarElevador`, `salirElevador` y `partidaElevador` para la clase `Elevador`. En las líneas 6 a 10 de la figura 10.30 se declaran los campos para estos atributos, y en las líneas 19 a 23 se declaran los métodos para estas operaciones. Las referencias `botonElevador`, `puertaElevador` y `timbre` (líneas 11 a 13) son campos especificados de las agregaciones de `Elevador` en la figura 10.28.

```
1  // Elevador.java
2  // Se generó utilizando los diagramas de clases 10.28 y 10.29
3  public class Elevador extends Ubicacion {
4
5     // atributos
6     private boolean movimiento;
7     private boolean llamado;
8     private Ubicacion pisoActual;
9     private Ubicacion pisoDestino;
10    private int tiempoRecorrido = 5;
11    private Boton botonElevador;
12    private Puerta puertaElevador;
13    private Timbre timbre;
14
15    // constructor
16    public Elevador() {}
17
```

Figura 10.30 La clase `Elevador` generada en base a las figuras 10.28 y 10.29. (Parte 1 de 2.)

```
18    // operaciones
19    public void viajar() {}
20    public void solicitarElevador() {}
21    public void entrarElevador() {}
22    public void salirElevador() {}
23    public void partida Elevador() {}
24
25    // método que sobrescribe a obtenerBoton
26    public Boton obtenerBoton()
27    {
28        return botonElevador;
29    }
30
31    // método que sobrescribe a obtenerPuerta
32    public Puerta obtenerPuerta()
33    {
34        return puertaElevador;
35    }
36 }
```

Figura 10.30 La clase `Elevador` generada en base a las figuras 10.28 y 10.29. (Parte 2 de 2.)

Tip para prevenir errores 10.1

Utilizar las herramientas de modelado de UML para generar código ayuda automáticamente a reducir la cantidad de errores de programación que los programadores tienden a introducir cuando escriben código en forma manual.

Observación de ingeniería de software 10.15

Varias herramientas de modelado de UML convierten a Java los diseños basados en UML. Estas herramientas pueden agilizar considerablemente el proceso de implementación.[3]

Le hemos proporcionado unas bases sólidas para implementar en Java diseños basados en UML. En la sección 11.9 regresaremos a las interacciones, con un enfoque en la manera en que los objetos generan y manejan los mensajes que se pasan en las colaboraciones, y aplicaremos ingeniería hacia delante a más diagramas de clases para convertirlos en código de Java. En los apéndices D, E y F presentaremos la implementación completa del código en Java para nuestro simulador.

10.12 (Opcional) Descubrimiento de patrones de diseño: Introducción a los patrones de diseño de creación, estructura y comportamiento

Ahora que hemos introducido la programación orientada a objetos, empezaremos nuestra presentación más detallada de los patrones de diseño. En la sección 1.16 mencionamos que la "Banda de los cuatro" describió 23 patrones de diseño utilizando tres categorías: de diseño, de estructura y de comportamiento. En ésta y las siguientes secciones "Descubrimiento de patrones de diseño" hablaremos sobre los patrones de diseño en cada categoría y su importancia, y de cómo cada patrón se relaciona con el material de Java en el libro. Por ejemplo, varios componentes Swing de Java que presentaremos en los capítulos 13 y 14 utilizan el patrón de diseño Compuesto, por lo que en la sección 10.12 presentamos este patrón de diseño. En la figura 10.31 se identifican los 18 patrones de diseño de la Banda de los cuatro que se describen en este libro.

En la figura 10.31 se muestran 18 de los patrones que se utilizan con mayor frecuencia en la industria de la ingeniería de software. Hay muchos patrones populares que se han documentado desde el libro de la Banda de los cuatro; entre éstos se incluyen los *patrones de diseño concurrentes*, que son especialmente útiles en el

3. Para obtener más información acerca de estas herramientas, consulte los recursos de Internet y World Wide Web que se muestran al final de la sección 2.9.

diseño de sistemas con subprocesamiento múltiple. Los patrones de arquitectura, como veremos en la sección 18.12, especifican la manera en que los subsistemas interactúan entre sí. En la figura 10.32 se muestran los patrones de concurrencia y los patrones de arquitectura que veremos en este libro.

10.12.1 Patrones de diseño de creación

Los *patrones de diseño de creación* tratan con las cuestiones relacionadas con la creación de los objetos, como evitar que un sistema cree más de un objeto de una clase (el patrón de diseño de creación Singleton) o diferir hasta el tiempo de ejecución la decisión acerca de qué tipos de objetos se van a crear (el propósito de los demás patrones de diseño de creación que describiremos aquí). Por ejemplo, suponga que vamos a diseñar un programa de dibujo en 3-D, en el cual el usuario puede crear varios objetos geométricos de 3-D como cilindros, esferas, cubos, tetraedros, etcétera. Además, suponga que cada figura en el programa de dibujo está representada por un objeto. En tiempo de compilación, el programa no sabe qué figuras el usuario elegirá para dibujar. Con base en la entrada del usuario, este programa debe ser capaz de determinar la clase de la cual debe instanciar un objeto apropiado para la figura seleccionada por el usuario. Si éste crea un cilindro en la GUI, nuestro programa debe "saber" cómo instanciar un objeto de la clase `Cilindro`. Cuando el usuario decide qué objeto geométrico dibujar, el programa debe determinar la subclase específica a partir de la cual va a instanciar ese objeto.

El libro de la Banda de los cuatro describe cinco patrones de creación (de los cuales describiremos cuatro en este libro):

- Fábrica abstracta (sección 18.12).
- Constructor (no se describe).
- Método de fábrica (sección 14.14).
- Prototipo (sección 22.12).
- Singleton (sección 10.12).

Sección	Patrones de diseño de creación	Patrones de diseño de estructura	Patrones de diseño de comportamiento
10.12	Singleton	Proxy	Memento, Estado
14.14	Método de fábrica	Adaptador, Puente, Compuesto	Cadena de responsabilidad, Comando, Observador, Estrategia, Método de plantilla
18.12	Fábrica abstracta	Decorador, Fachada	
22.12	Prototipo		Iterador

Figura 10.31 Los 18 patrones de diseño de la Banda de los cuatro, que se describen en *Cómo programar en Java, quinta edición.*

Sección	Patrones de diseño de concurrencia	Patrones de arquitectura
16.12	Ejecución de un solo subproceso, Suspensión precavida, Frustración, Bloqueo de lectura/escritura, Término en dos fases	
18.12		Modelo-vista-controlador, Niveles

Figura 10.32 Los patrones de diseño de concurrencia y patrones de arquitectura que se describen en *Cómo programar en Java, quinta edición.*

http://librosysolucionarios.net

Singleton

En ocasiones, un sistema debe contener exactamente un objeto de una clase (es decir, una vez que el programa crea una instancia de ese objeto, no se le debe permitir crear objetos adicionales de esa clase). Por ejemplo, algunos sistemas se conectan a una base de datos utilizando sólo un objeto que administre las conexiones de la base de datos, lo cual asegura que otros objetos no puedan inicializar conexiones innecesarias que reduzcan la velocidad del sistema. El *patrón de diseño Singleton* garantiza que un sistema cree, como máximo, la instancia de un objeto de una clase.

En la figura 10.33 se demuestra el código de Java utilizando el patrón de diseño Singleton. En la línea 4 se declara la clase `Singleton` como `final`, por lo que no pueden crearse subclases que proporcionen múltiples instanciamientos. En las líneas 10 a 13 se declara un constructor `private` (sólo la clase `Singleton` puede instanciar un objeto `Singleton` utilizando este constructor). En la línea 7 se declara una referencia estática a un objeto `Singleton` y se invoca al constructor privado. Esto crea la única instancia de la clase `Singleton` que se va a proporcionar a los clientes. Al ser invocado, el método estático `obtenerInstanciaSingleton` (líneas 16 a 19) simplemente devuelve una copia de esta referencia.

En las líneas 9 y 10 de la clase `PruebaSingleton` (figura 10.34) se declaran dos referencias a objetos `Singleton`: `primerSingleton` y `segundoSingleton`. En las líneas 13 y 14 se llama al método `obtenerInstanciaSingleton` y se asignan referencias `Singleton` a `primerSingleton` y `segundoSingleton`, respectivamente. En la línea 17 se prueba si ambas referencias se refieren al mismo objeto `Singleton`. En la figura 10.34 se muestra que, ciertamente, `primerSingleton` y `segundoSingleton` son ambas referencias al mismo objeto `Singleton`, ya que cada vez que se hace una llamada al método `obtenerInstanciaSingleton` se devuelve una referencia al mismo objeto `Singleton`.

```
1  // Singleton.java
2  // Demostración del patrón de diseño Singleton
3
4  public final class Singleton {
5
6     // objeto Singleton que será devuelto por obtenerInstanciaSingleton
7     private static final Singleton singleton = new Singleton();
8
9     // el constructor privado evita el instanciamiento por otros objetos
10    private Singleton()
11    {
12       System.err.println( "Objeto Singleton creado." );
13    }
14
15    // devolver objeto Singleton existente
16    public static Singleton obtenerInstanciaSingleton()
17    {
18       return singleton;
19    }
20 }
```

Figura 10.33 La clase `Singleton` asegura que sólo se cree un objeto de su clase.

```
1  // PruebaSingleton.java
2  // Intento de crear dos objetos Singleton
3
4  public class PruebaSingleton {
5
6     // ejecutar ejemplo Singleton
7     public static void main( String args[] )
8     {
9        Singleton primerSingleton;
10       Singleton segundoSingleton;
11
12       // crear objetos Singleton
13       primerSingleton = Singleton.obtenerInstanciaSingleton();
14       segundoSingleton = Singleton.obtenerInstanciaSingleton();
15
16       // las "dos" referencias Singleton deben referirse al mismo objeto
            Singleton
17       if ( primerSingleton == segundoSingleton )
18          System.err.println( "primerSingleton y segundoSingleton " +
19             "se refieren al mismo objeto Singleton" );
20    }
21 }
```

```
Objeto Singleton creado.
primerSingleton y segundoSingleton se refieren al mismo objeto Singleton.
```

Figura 10.34 La clase `PruebaSingleton` intenta crear un objeto `Singleton` más de una vez. (Parte 1 de 2.)

http://librosysolucionarios.net

Figura 10.34 La clase `PruebaSingleton` intenta crear un objeto `Singleton` más de una vez. (Parte 2 de 2.)

10.12.2 Patrones de diseño de estructura

Los *patrones de diseño de estructura* describen maneras comunes de organizar las clases y objetos en un sistema. El libro de la Banda de los cuatro describe siete patrones de diseño de estructura (seis de los cuales describiremos en este libro):

- Adaptador (sección 14.14).
- Puente (sección 14.14).
- Compuesto (sección 14.14).
- Decorador (sección 18.12).
- Fachada (sección 18.12).
- Flyweight (no se describe).
- Proxy (sección 10.12).

Proxy

Un applet debe siempre mostrar algo mientras las imágenes se cargan, para proporcionar una retroalimentación positiva a los usuarios, de manera que sepan que el applet está funcionando. Ya sea ese "algo" una imagen más pequeña o una cadena de texto que informe al usuario que las imágenes se están cargando, el *patrón de diseño Proxy* puede aplicarse para lograr este efecto. Este patrón permite a un objeto actuar como reemplazo para otro. Considere la carga de varias imágenes (varios megabytes) en un applet de Java. Idealmente, nos gustaría ver estas imágenes en forma instantánea —sin embargo, el proceso de cargar imágenes extensas en memoria puede requerir cierto tiempo para completarse (especialmente a través de una red). El patrón de diseño Proxy permite al sistema utilizar un objeto (conocido como *objeto proxy*) en lugar de otro. En nuestro ejemplo, el objeto proxy podría ser un medidor que informe al usuario cuál es el porcentaje que se ha cargado de una imagen extensa. Cuando esta imagen termine de cargarse, el objeto proxy ya no será necesario (el applet podrá entonces mostrar una imagen en vez del proxy). La clase `javax.swing.JProgressBar` puede utilizarse para crear este tipo de objetos proxy.

10.12.3 Patrones de diseño de comportamiento

Existen muchos ejemplos de *patrones de diseño de comportamiento*, los cuales proporcionan estrategias comprobadas para modelar la manera en que los objetos colaboran entre sí en un sistema, y ofrecen comportamientos especiales apropiados para una amplia variedad de aplicaciones. Analizaremos el patrón de diseño de comportamiento Observador (un clásico ejemplo de un patrón de diseño que muestra las colaboraciones entre los objetos). Por ejemplo, los componentes de la GUI colaboran con sus componentes de escucha para responder a las interacciones del usuario. Los componentes de la GUI utilizan este patrón para procesar los eventos de la interfaz de usuario. Un componente de escucha observa los cambios de estado en un componente específico de la GUI al registrarse para manejar los eventos de ese componente (de la GUI). Cuando el usuario interactúa con ese componente de la GUI, éste notifica a sus componentes de escucha (también conocidos como sus observadores) que el estado del componente de GUI ha cambiado (por ejemplo, cuando se oprime un botón).

También vamos a considerar el patrón de diseño de comportamiento Memento: un ejemplo de cómo ofrecer comportamiento especial para muchas aplicaciones. Este patrón permite a un sistema guardar el estado de un objeto, de manera que ese estado pueda restaurarse posteriormente. Por ejemplo, muchas aplicaciones proporcionan una capacidad de "deshacer", la cual permite a los usuarios regresar a versiones previas de su trabajo.

El libro de la Banda de los cuatro describe 11 patrones de diseño de comportamiento (ocho de los cuales describiremos en este libro):

- Cadena de responsabilidad (sección 14.14).
- Comando (sección 14.14).
- Intérprete (no se describe).
- Iterador (sección 22.12).
- Mediador (no se describe).
- Memento (sección 10.12).
- Observador (sección 14.14).
- Estado (sección 10.12).
- Estrategia (sección 14.14).
- Método de plantilla (sección 14.14).
- Visitante (no se describe).

Memento

Considere un programa de dibujo. Este tipo de programa permite a un usuario crear gráficos. Algunas veces el usuario puede colocar un gráfico en forma inapropiada en el área de dibujo. Los programas de dibujo ofrecen una característica de "deshacer", la cual permite al usuario corregir dicho error. Específicamente, el programa restaura el estado que tenía el área de dibujo antes de que el usuario colocara el gráfico. Algunos programas de dibujo más sofisticados incluyen un *historial*, el cual almacena varios estados en una lista para que el usuario pueda restaurar el programa a cualquier estado en el historial. El *patrón de diseño Memento* permite a un objeto guardar su estado, de manera que (si es necesario) este objeto pueda restaurarse a su estado anterior.

El patrón de diseño Memento requiere tres tipos de objetos. El *objeto originador* ocupa cierto *estado* (el conjunto de valores de atributos en un momento específico de la ejecución de un programa). En nuestro ejemplo del programa de dibujo, el área de dibujo actúa como el originador, ya que contiene la información del atributo que describe su estado; cuando el programa se ejecuta por primera vez, el área no contiene elementos. El *objeto memento* almacena una copia de los atributos necesarios asociados con el estado del originador (es decir, el memento almacena el estado del área de dibujo). El memento se almacena como el primer elemento en la lista del historial, la cual actúa como el *objeto cuidador* (el objeto que contiene referencias a todos los objetos memento asociados con el originador). Ahora, suponga que el usuario dibuja un círculo en el área de dibujo. Esta área contiene distinta información que describe su estado; un objeto círculo centrado en las coordenadas *x*-*y* especificadas. El área de dibujo usa entonces otro memento para almacenar esta información. Este memento se convierte en el segundo elemento en la lista del historial. Esta lista muestra todos los mementos en pantalla, de manera que el usuario pueda seleccionar cuál estado restaurar. Suponga que el usuario desea quitar el círculo; si el usuario selecciona el primer memento de la lista, el área de dibujo utiliza el primer memento para restaurar el área de dibujo en blanco.

Estado

En ciertos diseños debemos transmitir la información del estado de un objeto o representar los diversos estados que puede ocupar un objeto. Nuestro ejemplo práctico opcional de simulación del elevador en las secciones "Acerca de los objetos" utiliza el *patrón de diseño de Estado*. Nuestra simulación incluye un elevador que se mueve entre los pisos de un edificio de dos plantas. Una persona camina a través de un piso y viaja en el elevador hacia el otro piso. Originalmente utilizamos un valor entero para representar en qué piso está caminando la persona. Sin embargo, encontramos un problema al tratar de responder a la pregunta "¿en qué piso se encuentra la persona cuando viaja en el elevador?" En realidad, la persona no se encuentra en ningún piso, sino que está dentro del elevador. También nos dimos cuenta que el elevador y los pisos son ubicaciones que la persona puede ocupar en nuestra simulación. Creamos una superclase abstracta llamada `Ubicacion` para representar una "ubicación". Las subclases `Elevador` y `Piso` heredan de la superclase `Ubicacion`. La clase `Persona` contiene una referencia a un objeto `Ubicacion`, que representa la ubicación actual (elevador, primer piso o segundo piso) de esa persona. Como la referencia de una superclase puede guardar una referencia de subclase, el atributo `Ubicacion` de la persona hace referencia al objeto `Piso` apropiado cuando esa persona se encuentra en un piso, y hace referencia al objeto `Elevador` cuando esa persona está dentro del elevador.

El elevador y los pisos contienen botones. (El botón del elevador le indica que se mueva al otro piso, y los botones de los pisos llaman al elevador hacia el piso desde el cual se hizo la solicitud.) Como todas las ubicaciones en nuestra simulación contienen botones, la clase `Ubicacion` proporciona el método abstracto `obtenerBoton`. La clase `Elevador` implementa el método `obtenerBoton` para devolver una referencia al objeto `Boton` dentro del elevador, y la clase `Piso` implementa el método `obtenerBoton` para devolver una referencia al objeto `Boton` en el piso. Mediante el uso de su referencia `Ubicacion`, la persona puede oprimir el botón correcto (es decir, la persona no oprimirá un botón de piso cuando se encuentre dentro del elevador, y no oprimirá el botón del elevador cuando se encuentre en un piso).

El patrón de diseño de Estado utiliza una superclase abstracta (conocida como la *clase de Estado*), la cual contiene métodos que describen los comportamientos para los estados que un objeto (conocido como el *objeto de contexto*) puede ocupar. En nuestra simulación del elevador, la clase de Estado es la superclase `Ubicacion`, y el objeto de contexto es el objeto de la clase `Persona`. Observe que la clase `Ubicacion` no describe a todos los estados de la clase `Persona` (por ejemplo, si esa persona está caminando o si espera el elevador); la clase `Ubicacion` describe solamente la ubicación de la `Persona` y contiene el método `obtenerBoton` para que la `Persona` pueda acceder al objeto `Boton` en varias ubicaciones.

Una *subclase de Estado*, que extiende a la clase de Estado, representa un estado individual que el contexto puede ocupar. Las subclases de Estado en nuestra simulación son `Elevador` y `Piso`. Cada subclase de Estado contiene métodos que implementan a los métodos abstractos de la clase de Estado. Por ejemplo, ambas clases `Elevador` y `Piso` implementan el método `obtenerBoton`.

El contexto contiene exactamente una referencia a un objeto de la clase de Estado; este objeto se conoce como el *objeto de estado*. En la simulación, el objeto de estado es el objeto de la clase `Ubicacion`. Cuando el contexto cambia de estado, el objeto de estado hace referencia al objeto de la subclase de Estado asociado con ese nuevo estado. Por ejemplo, cuando la persona camina del piso hacia el elevador, la `Ubicacion` del objeto `Persona` cambia de hacer referencia a uno de los objetos `Piso`, a hacer referencia al objeto `Elevador`. Cuando la persona camina del elevador hacia el piso, la `Ubicación` del objeto `Persona` hace referencia al objeto `Piso` apropiado.

10.12.4 Conclusión

En la sección 10.12, "Descubrimiento de patrones de diseño", enlistamos los tres tipos de patrones de diseño introducidos en el libro de la Banda de los cuatro, identificamos 18 de estos patrones de diseño que veremos en este libro, y discutimos sobre ciertos patrones de diseño en especial, incluyendo a Singleton, Proxy, Memento y Estado. En la sección 14.14, "Descubrimiento de patrones de diseño", introduciremos algunos patrones de diseño asociados con los componentes de la GUI de AWT y Swing. Después de leer esta sección, usted debe comprender mejor la forma como los componentes de la GUI de Java aprovechan los patrones de diseño.

http://librosysolucionarios.net

10.12.5 Recursos en Internet y World Wide Web

Las siguientes direcciones URLs proporcionan más información acerca de la naturaleza, importancia y aplicaciones de los patrones de diseño.

Patrones de diseño
`www.hillside.net/patterns`
Esta página muestra vínculos a información sobre los patrones de diseño y los lenguajes.

`www.hillside.net/patterns/books/`
Este sitio muestra libros sobre patrones de diseño.

`www.netobjectives.com/design.htm`
Este sitio trata sobre la importancia de los patrones de diseño.

`umbc7.umbc.edu/~tarr/dp/dp.html`
Este sitio muestra vínculos a sitios Web sobre patrones de diseño tutoriales y artículos.

`www.c2.com/ppr/`
Este sitio habla sobre los avances recientes en los patrones de diseño e ideas para proyectos futuros.

Patrones de diseño en Java
`www.cmcrossroads.com/bradapp/javapats.html`
Este sitio habla sobre los patrones de diseño en Java y presenta los patrones de diseño en la computación distribuida.

Patrones de arquitectura
`www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html`
Este sitio contiene un artículo en el que se describe cómo los componentes de Swing utilizan la arquitectura Modelo-vista-controlador.

`www.ootips.org/mvc-pattern.html`
Este sitio proporciona información y tips acerca del uso de MVC.

`www.tml.hut.fi/Opinnot/Tik-109.450/1998/niska/sld001.htm`
Este sitio proporciona información sobre los patrones de arquitectura, el patrón de diseño y los modismos (patrones dirigidos a un lenguaje específico).

RESUMEN

- Con el polimorfismo es posible diseñar e implementar sistemas que puedan extenderse más fácilmente. Pueden escribirse programas para procesar objetos de tipos que tal vez no existan cuando el programa se encuentre en desarrollo.
- Existen muchas situaciones en las que es conveniente declarar clases `abstract`, para las cuales el programador no tenga la intención de crear objetos. Como estas clases se utilizan sólo como superclases, nos referimos a ellas como superclases abstractas. No pueden crearse objetos de una clase `abstract`.
- Las clases a partir de las cuales se crean objetos, se conocen como clases concretas.
- Una clase debe declararse como `abstract` si uno o más de sus métodos son `abstract`. Un método abstracto es un método con la palabra clave `abstract` en su declaración.
- Si una clase extiende a otra clase con un método `abstract` y no implementa a ese método `abstract`, entonces ese método seguirá siendo `abstract` en la subclase. Por consecuencia, la subclase será también una clase `abstract`, y deberá declararse como `abstract`.
- Java permite el polimorfismo (la habilidad de que objetos de distintas clases, relacionadas por la herencia o la implementación de una interfaz, respondan de manera distinta a la llamada a un mismo método).
- Cuando se hace una solicitud a través de la referencia de una superclase para usar un método `abstract`, Java selecciona el método sobrescrito correcto en la subclase del objeto.
- Aunque no podemos instanciar objetos de clases `abstract`, sí podemos declarar variables de tipos de clases abstractas. Dichas variables pueden usarse para hacer referencia a objetos de subclases.
- Con frecuencia se agregan nuevos tipos de clases a los sistemas. Estas nuevas clases se acomodan mediante la vinculación dinámica (también conocida como vinculación postergada). No necesita conocerse el tipo de un objeto en tiempo de compilación para que se pueda compilar la llamada a un método. En tiempo de ejecución, el método apropiado será llamado en el objeto al que se refiera la referencia.

- El operador `instanceof` comprueba el tipo del objeto al que se refiere su operando izquierdo, y determina si este tipo tiene una relación *es un* con el tipo especificado como su operando derecho. Si los dos tienen una relación *es un*, `instanceof` devuelve `true`. De no ser así, `instanceof` devuelve `false`.
- Todo objeto en Java conoce su propia clase y puede acceder a esta información mediante el método `getClass`, el cual todas las clases heredan de la clase `Object`. El método `getClass` devuelve un objeto de tipo `Class` (paquete `java.lang`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase. El método `getName` de `Class` devuelve el nombre de la clase.
- La declaración de una interfaz empieza con la palabra clave `interface` y contiene un conjunto de métodos `public abstract`. Las interfaces pueden también contener campos `public static final`.
- Para utilizar una interfaz, una clase debe especificar que la va a implementar (mediante la palabra clave `implements`) y debe declarar cada método en la interfaz con las firmas especificadas en la declaración de la misma.
- Una interfaz se utiliza generalmente en lugar de una clase `abstract` cuando no hay una implementación predeterminada que heredar.
- Cuando una clase implementa a una interfaz, establece una relación "es un" con el tipo de la interfaz.
- Para implementar más de una interfaz, simplemente debe proporcionar una lista de nombres de interfaces separados por comas después de la palabra clave `implements` en la declaración de la clase.
- Las clases anidadas se declaran dentro de otras clases.
- Una clase interna tiene acceso a todas las variables y métodos de su clase externa.
- Una clase interna puede declararse dentro de un método. Dicha clase interna tiene acceso a los miembros de su clase externa y a las variables locales `final` del método en el que está declarada.
- Las clases internas se utilizan principalmente en el manejo de eventos.
- Una clase interna anónima no tiene nombre y se declara en el punto del programa en el que se crea un objeto de esa clase.
- Una clase interna anónima puede implementar una interfaz o extender a una clase.
- Al compilar una clase que contiene clases anidadas se produce un archivo `.class` separado por cada clase declarada.
- Las clases internas pueden declararse como `public`, `protected`, con acceso a nivel de paquete o `private`, y están sujetas a las mismas restricciones de uso que los demás miembros de una clase.
- Para acceder a la referencia `this` de la clase externa en una clase interna, use *NombreClaseExterna*`.this`.
- Una clase interna es una clase anidada que no se declara como `static`.
- La clase `JFrame` proporciona los atributos y comportamientos básicos de una ventana: una barra de título y botones para minimizar, maximizar y cerrar la ventana.
- El evento generado cuando el usuario hace clic en el cuadro para cerrar la ventana es un evento de cierre de ventana.
- El método `addWindowListener` registra un componente de escucha de eventos de ventana. El argumento para `addWindowListener` debe ser una referencia a un objeto que *sea un* `WindowListener`.
- Para las interfaces de manejo de eventos con más de un método, Java proporciona una clase adaptadora correspondiente que ya implementa a todos los métodos en la interfaz por usted. La clase `WindowAdapter` implementa a la interfaz `WindowListener`, de manera que cada objeto `WindowAdapter` *es un* `WindowListener`.

TERMINOLOGÍA

`abstract`, clase
`abstract`, método
`abstract`, palabra clave
`abstract`, superclase
`Boolean`, clase
`Byte`, clase
`Character`, clase
clase anidada
clase concreta
clase de envoltura de tipos
clase interna
clase interna anónima
`Class`, clase
constante en una interfaz
conversión descendente (downcasting)
`Double`, clase
extensibilidad
`final`, clase
`final`, método
`final`, variable de instancia
`Float`, clase
`getClass`, método de `Object`
`getName`, método de `Class`
herencia de implementación
herencia de interfaz
implementar una interfaz
`implements`, palabra clave
`instanceof`, operador
`Integer`, clase
`interface`, palabra clave
interfaz
jerarquía de clases
`JFrame`, clase
`Long`, clase
método `abstract` de una superclase
`Number`, clase
polimorfismo
programación "en forma específica"
programación "en forma general"
referencia
referencia a una clase `abstract`
`setDefaultCloseOperation`, método de `JFrame`
`setSize`, método de la clase `Component`
`setVisible`, método de la clase `Component`
`Short`, clase
sobrescribir un método `abstract`
tipo de referencia
variable de un tipo de superclase

http://librosysolucionarios.net

vinculación dinámica de métodos
vinculación estática
vinculación postergada
`WindowAdapter`, clase
`windowClosing`, método de `WindowListener`
`WindowEvent`, clase
`WindowListener`, interfaz

EJERCICIOS DE AUTOEVALUACIÓN

10.1 Complete las siguientes oraciones:

a) Tratar al objeto de una superclase como un(a) _____ puede producir errores.
b) El polimorfismo ayuda a eliminar la lógica de _____.
c) Si una clase contiene al menos un método abstracto, es una clase _____.
d) Las clases a partir de las cuales pueden instanciarse objetos se llaman clases _____.
e) El _____ implica el uso de una variable de superclase para invocar métodos en objetos de superclase y subclase.
f) Los métodos abstractos se declaran utilizando la palabra clave _____.
g) Al proceso de convertir un objeto de superclase a un objeto de subclase se le conoce como _____.

10.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Es posible tratar a los objetos de superclase y a los objetos de subclase de manera similar.
b) Todos los métodos en una clase `abstract` deben declararse como métodos `abstract`.
c) Es peligroso referirse a un objeto de subclase con una variable de superclase.
d) Una clase se hace abstracta declarándola como `abstract`.
e) Si una superclase declara a un método como `abstract`, una subclase debe implementar a ese método para convertirse en clase concreta.
f) Las clases internas no pueden acceder a los miembros de la clase circundante.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

10.1 a) Objeto de subclase. b) `switch`. c) Abstractas. d) Concretas. e) Polimorfismo. f) `abstract`. g) Conversión descendente (downcasting).

10.2 a) Verdadero. b) Falso. Una clase abstracta puede incluir métodos con implementaciones. c) Falso. Es peligroso referirse a un objeto de superclase con una variable de subclase. d) Verdadero. e) Verdadero. f) Falso. Las clases internas tienen acceso a todos los miembros de la declaración de clase circundante.

EJERCICIOS

10.3 ¿Cómo es que el polimorfismo le permite programar "en forma general", en lugar de hacerlo "en forma específica"? Hable sobre las ventajas clave de la programación "en forma general".

10.4 Indique la diferencia entre heredar la interfaz y heredar la implementación. ¿En qué difieren las jerarquías diseñadas para heredar la interfaz de las jerarquía diseñadas para heredar la implementación?

10.5 ¿Qué son los métodos `abstract`? Describa una circunstancia en la que los métodos `abstractos` serían apropiados.

10.6 ¿Cómo es que el polimorfismo fomenta la extensibilidad?

10.7 Modifique el sistema de nómina de las figuras 10.13 a 10.17 para incluir la variable de instancia `private` llamada `fechaNacimiento` (use la clase `Fecha`) en la clase `Empleado`. Suponga que la nómina se procesa una vez al mes. Cree un arreglo de variables `Empleado` para guardar referencias a los diversos objetos empleado. En un ciclo, calcule la nómina para cada `Empleado` (mediante el polimorfismo) y agregue una bonificación de $100.00 a la cantidad de pago de nómina de la persona, si el mes actual es el mes en el que ocurre el cumpleaños de ese `Empleado`.

10.8 Implemente la jerarquía `Figura` que se muestra en la figura 9.3. Cada `FiguraBidimensional` debe contener el método `obtenerArea` para calcular el área de la figura bidimensional. Cada `FiguraTridimensional` debe tener los métodos `obtenerArea` y `obtenerVolumen` para calcular el área superficial y el volumen, respectivamente, de la figura tridimensional. Cree un programa que utilice un arreglo de referencias `Figura` a objetos de cada clase concreta en la jerarquía. El programa deberá imprimir el objeto al cual se refiere cada elemento del arreglo. Además, en el ciclo que procesa a todas las figuras en el arreglo, determine si cada figura es `FiguraBidimensional` o `FiguraTridimensional`. Si es `FiguraBidimensional`, muestre su área. Si es `FiguraTridimensional`, muestre su área y su volumen.

10.9 *(Aplicación de dibujo.)* Modifique el programa de dibujo del ejercicio 8.18 para crear una aplicación que dibuje líneas, rectángulos y óvalos en forma aleatoria. (*Nota:* Al igual que un applet, un objeto `JFrame` tiene un método `paint` que usted puede sobrescribir para dibujar en el fondo del `JFrame`.)

Para este ejercicio, modifique las clases `MiLinea`, `MiOvalo` y `MiRect` del ejercicio 8.18 para crear la jerarquía de clases de la figura 10.35. Las clases de la jerarquía `MiFigura` deben ser clases de figuras "inteligentes", de tal manera que los objetos de estas clases sepan cómo dibujarse a sí mismos (si se les proporciona un objeto `Graphics` que les indique en dónde dibujar). La única lógica `switch` o `if...else` en su programa deberá determinar el tipo del objeto figura que se va a crear. (Use números aleatorios para elegir el tipo de figura y sus coordenadas.) Una vez creado un objeto de esta jerarquía, se manipulará por el resto de su vida útil como una referencia a la superclase `MiFigura`.

La clase `MiFigura` en la figura 10.35 *debe* ser abstracta. Los únicos datos que representan las coordenadas de las figuras en la jerarquía deben declararse en la clase `MiFigura`. Las líneas, rectángulos y óvalos pueden dibujarse si se conocen dos puntos en el espacio. Las líneas requieren las coordenadas *x1*, *y1*, *x2* y *y2*. El método `drawLine` de la clase `Graphics` conecta los dos puntos suministrados con una línea. Si usted tiene los mismos cuatro valores de coordenada (*x1*, *y1*, *x2* y *y2*) para óvalos y rectángulos, puede calcular los cuatro argumentos necesarios para dibujarlos. Cada uno requiere de un valor de coordenada *x* superior izquierda (el mínimo de los dos valores de coordenada *x*), un valor de coordenada *y* superior izquierda (el mínimo de los dos valores de coordenada *y*), una *anchura* (el valor absoluto de la diferencia entre los dos valores de coordenada *x*) y una *altura* (el valor absoluto de la diferencia entre los dos valores de coordenada *y*). [*Nota:* En el capítulo 13, cada par *x,y* se captura mediante el uso de eventos del ratón, provenientes de las interacciones entre el usuario y el fondo del programa. Estas coordenadas se almacenan en un objeto figura apropiado seleccionado por el usuario. Al empezar el ejercicio, usted utilizará valores de coordenadas aleatorios como argumentos para el constructor.]

Además de los datos comunes, la clase `MiFigura` debe declarar cuando menos los siguientes métodos:

a) Un constructor sin argumentos que establezca las coordenadas en 0.
b) Un constructor con argumentos que establezca las coordenadas en los valores suministrados.
c) Métodos establecer, para cada pieza individual de datos, que permitan al programador establecer cualquier pieza de datos de manera independiente para una figura en la jerarquía (por ejemplo, si tiene una variable de instancia `x1`, debe tener un método `establecerX1`).
d) Métodos obtener, para cada pieza individual de datos, que permitan al programador recuperar cualquier pieza de datos de manera independiente para una figura en la jerarquía (por ejemplo, si tiene la variable de instancia `x1`, debe tener un método `obtenerX1`).
e) El método abstracto

```
public abstract void dibujar( Graphics g );
```

que será llamado desde el método `paint` del programa para dibujar una figura en la pantalla.

Los métodos anteriores son requeridos. Si desea proporcionar más métodos para aumentar la flexibilidad, puede hacerlo. Sin embargo, asegúrese de que cualquier método que declare en esta clase sea un método utilizado por *todas* las figuras en la jerarquía.

Todos los datos *deben* ser privados para la clase `MiFigura`. Esto lo obligará a utilizar un encapsulamiento apropiado de los datos, junto con métodos *establecer/obtener* para manipular esos datos. Usted no podrá declarar nuevos datos que puedan derivarse de la información existente. Como se explicó anteriormente, la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la *anchura* y la *altura* necesarias para dibujar un óvalo o un

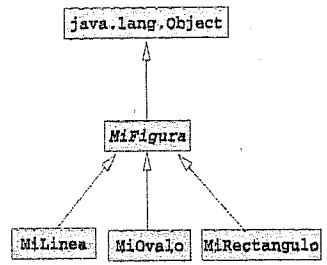


Figura 10.35 La jerarquía de `MiFigura`.

http://librosysolucionarios.net

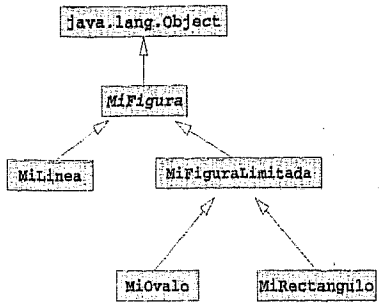


Figura 10.36 La jerarquía de `MiFigura` con `MiFiguraLimitada`.

rectángulo pueden calcularse si usted ya conoce dos puntos en el espacio. Todas las subclases de `MiFigura` deberán proporcionar dos constructores que imiten a los que se proporcionan mediante la clase `MiFigura`.

Los objetos de las clases `MiOvalo` y `MiRect` no deben calcular su coordenada *x* superior izquierda, coordenada *y* superior izquierda, *anchura* o *altura* sino hasta que estén a punto de dibujarse. Nunca deberá modificar las coordenadas *x1*, *y1*, *x2* y *y2* de un objeto `MiOvalo` o `MiRect` al prepararse para dibujarlos. En vez de ello, deberá utilizar los resultados temporales de los cálculos antes descritos. Esto nos ayudará a mejorar el programa en el capítulo 13, al permitir que el usuario seleccione las coordenadas de cada figura con el ratón.

No debe haber variables `MiLinea`, `MiOvalo` ni `MiRect` en el programa; sólo variables `MiFigura` que contengan referencias a objetos `MiLinea`, `MiOvalo` y `MiRect`. El programa deberá mantener un arreglo de variables `MiFigura` que contenga todas las figuras. El método `paint` del programa deberá recorrer el arreglo de variables `MiFigura` y dibujar cada figura (es decir, llamar al método `dibujar` de cada figura).

Empiece por declarar la clase `MiFigura`, la clase `MiLinea` y una aplicación para probar sus clases. La aplicación deberá tener una variable de instancia `MiFigura` que pueda hacer referencia a un objeto `MiLinea` (creado en el constructor de la aplicación). El método `paint` (para su subclase de `JFrame`) deberá dibujar la figura mediante una instrucción como la siguiente:

```
figuraActual.dibujar( g );
```

en donde `figuraActual` es la referencia `MiFigura` y `g` es el objeto `Graphics` que será utilizado por la figura para dibujarse a sí misma en el fondo de la ventana.

A continuación, cambie la referencia `MiFigura` por un arreglo de referencias `MiFigura`, y codifique directamente en el programa varios objetos `MiLinea` para dibujarlos. El método `paint` de la aplicación deberá recorrer el arreglo de figuras y dibujar cada una de ellas.

Una vez que la parte anterior esté funcionando, deberá declarar las clases `MiOvalo` y `MiRect` y agregar objetos de esas clases en el arreglo existente. Por ahora, todos los objetos figura se crearán en el constructor de su subclase de `JFrame`. En el capítulo 13 crearemos los objetos cuando el usuario seleccione una figura y empiece a dibujarla con el ratón.

10.10 En el ejercicio 10.9, usted creó una jerarquía `MiFigura` en la que las clases `MiLinea`, `MiOvalo` y `MiRect` extienden directamente a `MiFigura`. Si la jerarquía se diseñó apropiadamente, usted deberá ser capaz de ver las tremendas similitudes entre las clases `MiOvalo` y `MiRect`. Rediseñe y reimplemente el código de las clases `MiOvalo` y `MiRect` para "factorizar" las características comunes en la clase abstracta `MiFiguraLimitada`, para producir la jerarquía de la figura 10.36.

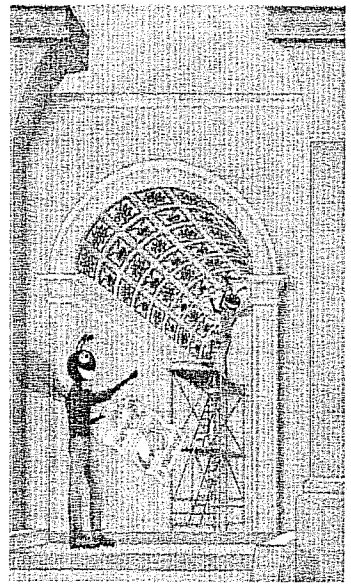
La clase `MiFiguraLimitada` debe declarar dos constructores que imiten a los constructores de la clase `MiFigura`, y debe también declarar métodos que calculen la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la *anchura* y la *altura*. No deben declararse en esta clase datos relacionados con las dimensiones de las figuras. Recuerde que los valores necesarios para dibujar un óvalo o rectángulo pueden calcularse a partir de dos coordenadas (*x*,*y*). Si se diseñan apropiadamente, las nuevas clases `MiOvalo` y `MiRect` deberán tener, cada una, dos constructores y un método `dibujar`.

11

Cadenas y caracteres

Objetivos

- Crear y manipular objetos de cadenas de caracteres no modificables, de la clase `String`.
- Crear y manipular objetos de cadenas de caracteres modificables, de la clase `StringBuffer`.
- Crear y manipular objetos de la clase `Character`.
- Utilizar un objeto `StringTokenizer` para descomponer un objeto `String` en tokens.



El principal defecto del rey Enrique
era masticar pequeños pedazos de hilo.
Hilaire Belloc

La buena escritura debe ser concisa. Una oración no
debe contener palabras innecesarias, un párrafo
no debe contener oraciones innecesarias.
William Strunk, Jr.

He extendido esta carta más de lo usual, ya que no
tengo el tiempo necesario para hacerla breve.
Blaise Pascal

La diferencia entre la palabra casi correcta y la palabra
correcta es realmente enorme; es como la diferencia
entre la luciérnaga y el rayo.
Mark Twain

Ésa es la palabra.
Miguel de Cervantes

http://librosysolucionarios.net

Plan general

11.1 Introducción
11.2 Fundamentos de los caracteres y las cadenas
11.3 La clase `String`
11.3.1 Constructores de `String`
11.3.2 Métodos `length`, `charArt` y `getChars` de `String`
11.3.3 Comparación entre cadenas
11.3.4 Localización de caracteres y subcadenas en las cadenas
11.3.5 Cómo extraer subcadenas de las cadenas
11.3.6 Concatenación de cadenas
11.3.7 Métodos varios de `String`
11.3.8 El método `valueOf` de `String`
11.4 La clase `StringBuffer`
11.4.1 Constructores de `StringBuffer`
11.4.2 Los métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuffer`
11.4.3 Los métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuffer`
11.4.4 Los métodos `append` de `StringBuffer`
11.4.5 Los métodos `insert` y `delete` de `StringBuffer`
11.5 La clase `Character`
11.6 La clase `StringTokenizer`
11.7 Simulación para barajar y repartir cartas
11.8 Expresiones regulares, la clase `Pattern` y la clase `Matcher`
11.9 (Ejemplo práctico opcional) Acerca de los objetos: Manejo de eventos

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Sección especial: Ejercicios avanzados de manipulación de cadenas • Sección especial: Proyectos desafiantes de manipulación de cadenas

11.1 Introducción

En este capítulo presentamos las herramientas para procesamiento de cadenas y caracteres en Java. Las técnicas que se describen aquí son apropiadas para validar la entrada en los programas, para mostrar información a los usuarios y otras manipulaciones basadas en texto. Estas técnicas son también apropiadas para desarrollar editores de texto, procesadores de palabras, software para diseño de páginas, sistemas de composición computarizados y demás tipos de software para procesamiento de texto. Ya hemos presentado varias herramientas para procesamiento de texto en capítulos anteriores. En este capítulo describiremos detalladamente las herramientas de las clases ***`String`***, ***`StringBuffer`*** y ***`Character`*** del paquete **`java.lang`**, y la clase ***`StringTokenizer`*** del paquete **`java.util`**. Estas clases proporcionan la base para la manipulación de cadenas y caracteres en Java.

11.2 Fundamentos de los caracteres y las cadenas

Los caracteres son los bloques de construcción fundamentales de los programas fuente de Java. Todo programa está compuesto de una secuencia de caracteres que, cuando se agrupan en forma significativa, son interpretados por la computadora como una serie de instrucciones utilizadas para realizar una tarea. Un programa puede contener *literales de carácter*. Una literal de carácter es un valor entero representado como un carácter entre comillas simples. Por ejemplo, **`'z'`** representa el valor entero de **`z`**, y **`'\n'`** representa el valor entero de una nueva línea. El valor de una literal de carácter es el valor entero del carácter en el *conjunto de caracteres Unicode*.[1]

Recuerde que en la sección 2.2 vimos que una cadena es una secuencia de caracteres que se tratan como una sola unidad; y puede incluir letras, dígitos y varios *caracteres especiales*, como **`+`**, **`-`**, **`*`**, **`/`** y **`$`**. Una cadena es un objeto de la clase **`String`**. Las *literales de cadena* (que se almacenan en memoria como *objetos `String` anónimos*) se escriben como una secuencia de caracteres entre comillas dobles, como en:

`"Juan E. Pérez"`	(un nombre)
`"Calle Principal 9999"`	(una dirección)
`"Monterrey, Nuevo León"`	(una ciudad y un estado)
`"(201) 555-1212"`	(un número telefónico)

Una cadena puede asignarse en una declaración a una referencia **`String`**. La declaración:

```
String color = "azul";
```

inicializa la referencia **`String`** de nombre **`color`** para que haga referencia a un objeto **`String`** anónimo que contiene la cadena **`"azul"`**.



Tip de rendimiento 11.1

*Java trata a todas las literales de cadena con el mismo contenido como un objeto **`String`** anónimo que tiene muchas referencias. Esto ayuda a conservar memoria.*

11.3 La clase `String`

La clase **`String`** se utiliza para representar cadenas en Java. En las siguientes subsecciones cubriremos muchas de las herramientas de la clase **`String`**.

11.3.1 Constructores de `String`

La clase **`String`** proporciona nueve constructores para inicializar objetos **`String`** de varias formas. Seis de los constructores se demuestran en el método **`main`** de la figura 11.1.

En la línea 17 se crea una instancia de un nuevo objeto **`String`**, utilizando el constructor predeterminado de **`String`**, y se le asigna su referencia a **`s1`**. El nuevo objeto **`String`** no contiene caracteres (la *cadena vacía*) y tiene una longitud de 0.

En la línea 18 se crea una instancia de un nuevo objeto **`String`**, utilizando el constructor de **`String`** que toma un objeto **`String`** como argumento, y se le asigna su referencia a **`s2`**. El nuevo objeto **`String`** contiene la misma secuencia de caracteres que el objeto **`String`** de nombre **`s`**, el cual se pasa como argumento para el constructor.

```
1  // Fig. 11.1: ConstructoresString.java
2  // Constructores de la clase String.
3  import javax.swing.*;
4
5  public class ConstructoresString {
6
7     public static void main( String args[] )
8     {
```

Figura 11.1 Constructores de la clase **`String`**. (Parte 1 de 2.)

1. En el apéndice B se presentan los equivalentes enteros de los caracteres en el conjunto de caracteres ASCII, el cual es un subconjunto de Unicode. En el apéndice G hablaremos sobre el conjunto de caracteres Unicode. Para obtener información detallada sobre Unicode, visite el sitio **`www.unicode.org`**.

http://librosysolucionarios.net

```
9        char arregloChar[] = { 'c', 'u', 'm', 'p', 'l', 'e', ' ', 'a', 'ñ', 'o', 's' };
10       byte arregloByte[] = { ( byte ) 'a', ( byte ) 'ñ',
11          ( byte ) 'o', ( byte ) ' ', ( byte ) 'n',
12          ( byte ) 'u', ( byte ) 'e', ( byte ) 'v', (byte) 'o' };
13
14       String s = new String( "hola" );
15
16       // usar constructores de String
17       String s1 = new String();
18       String s2 = new String( s );
19       String s3 = new String( arregloChar );
20       String s4 = new String( arregloChar, 7, 4 );
21       String s5 = new String( arregloByte, 4, 5 );
22       String s6 = new String( arregloByte );
23
24       // anexar objetos String a salida
25       String salida = "s1 = " + s1 + "\ns2 = " + s2 + "\ns3 = " + s3 +
26          "\ns4 = " + s4 + "\ns5 = " + s5 + "\ns6 = " + s6;
27
28       JOptionPane.showMessageDialog( null, salida,
29          "Constructores de la clase String", JOptionPane.INFORMATION_MESSAGE );
30
31       System.exit( 0 );
32    }
33
34 } // fin de la clase ConstructoresString
```

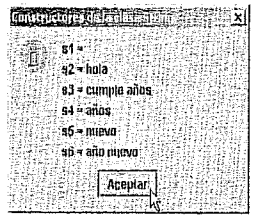


Figura 11.1 Constructores de la clase `String`. (Parte 2 de 2.)

Observación de ingeniería de software 11.1

No es necesario copiar un objeto `String` existente. Los objetos `String` son inmutables; su contenido de caracteres no puede modificarse una vez que son creados. Si hay una o más referencias a un objeto `String` (o a cualquier objeto), el objeto no puede ser reclamado por el recolector de basura. Por lo tanto, no puede utilizarse una referencia `String` para modificar un objeto `String`, o para eliminarlo de memoria, como en otros lenguajes de programación como C o C++.

En la línea 19 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s3`, utilizando el constructor de la clase `String` que toma un arreglo de caracteres como argumento. El nuevo objeto `String` contiene una copia de los caracteres en el arreglo.

En la línea 20 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s4`, utilizando el constructor de la clase `String` que toma un arreglo `char` y dos enteros como argumentos. El segundo argumento especifica la posición inicial (el *`desplazamiento`*) a partir del cual se accede a los caracteres en el arreglo. Recuerde que el primer carácter se encuentra en la posición 0. El tercer argumento especifica el número de caracteres (la *`cuenta`*) que se van a utilizar del arreglo. El nuevo objeto `String` contiene una cadena formada a partir de los caracteres utilizados. Si el `desplazamiento` o la `cuenta` especificados como

argumentos ocasionan que se acceda a un elemento fuera de los límites del arreglo de caracteres, se lanza una excepción `StringIndexOutOfBoundsException`. Hablaremos detalladamente sobre las excepciones en el capítulo 15.

Error común de programación 11.1

Intentar acceder a un carácter que se encuentra fuera de los límites de una cadena (es decir, un índice menor que 0 o un índice mayor o igual a la longitud de la cadena), se produce una excepción `StringIndexOutOfBoundsException`.

En la línea 21 se crea la instancia de un nuevo objeto `String` y se asigna su referencia a `s5`, utilizando el constructor de la clase `String` que toma un arreglo `byte` y dos enteros como argumentos. Los argumentos segundo y tercero especifican el `desplazamiento` y la `cuenta`, respectivamente. El nuevo objeto `String` contiene copias de los bytes especificados. Si el `desplazamiento` o la `cuenta` que se especifican como argumentos provocan el acceso a un elemento fuera de los límites del arreglo, se lanza una excepción `StringIndexOutOfBoundsException`.

En la línea 22 se crea la instancia de un nuevo objeto `String` y se le asigna la referencia `s6`, utilizando el constructor de la clase `String` que toma un arreglo `byte` como argumento. El nuevo objeto `String` contiene copias de los bytes en el arreglo.

11.3.2 Métodos `length`, `charArt` y `getChars` de `String`

Los métodos *`length`*, *`charAt`*, y *`getChars`* de `String` determinan la longitud de una cadena, obtienen el carácter que se encuentra en una ubicación específica de una cadena y recuperan el conjunto completo de caracteres en una cadena, respectivamente. La aplicación de la figura 11.2 demuestra a cada uno de estos métodos.

```
1  // Fig. 11.2: VariosString.java
2  // Este programa demuestra los métodos length, charAt y getChars
3  // de la clase String.
4  import javax.swing.*;
5
6  public class VariosString {
7
8     public static void main( String args[] )
9     {
10       String s1 = "hola a todos";
11       char arregloChar[] = new char[ 4 ];
12
13       String salida = "s1: " + s1;
14
15       // probar método length
16       salida += "\nLongitud de s1: " + s1.length();
17
18       // iterar a través de los caracteres en s1 y mostrar al revés
19       salida += "\nLa cadena invertida es: ";
20
21       for ( int cuenta = s1.length() - 1; cuenta >= 0; cuenta-- )
22          salida += s1.charAt( cuenta ) + " ";
23
24       // copiar caracteres de la cadena a arregloChar
25       s1.getChars( 0, 4, arregloChar, 0 );
26       salida += "\nEl arreglo de caracteres es: ";
27
28       for ( int cuenta = 0; cuenta < arregloChar.length; cuenta++ )
29          salida += arregloChar[ cuenta ];
```

Figura 11.2 Los métodos de manipulación de caracteres de la clase `String`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
30
31          JOptionPane.showMessageDialog( null, salida,
32             "Métodos de la clase String para manipular caracteres",
33             JOptionPane.INFORMATION_MESSAGE );
34
35          System.exit( 0 );
36       }
37
38    } // fin de la clase VariosString
```

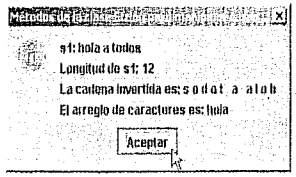


Figura 11.2 Los métodos de manipulación de caracteres de la clase `String`. (Parte 2 de 2.)

En la línea 16 se utiliza el método `length` de `String` para determinar el número de caracteres en la cadena `s1`. Al igual que los arreglos, las cadenas siempre conocen su propia longitud. Sin embargo, a diferencia de los arreglos, las cadenas no tienen un campo `length` que guarde el número de elementos en una cadena.

La instrucción `for` de las líneas 21 y 22 anexa a la cadena `salida` los caracteres de la cadena `s1` en orden inverso (y separados por espacios). El método `charAt` de `String` (línea 22) devuelve el carácter ubicado en una posición específica en la cadena. El método `charAt` recibe un argumento entero que se utiliza como el *índice*, y devuelve el carácter en esa posición. Al igual que los arreglos, se considera que el primer elemento de una cadena está en la posición 0.

Tip para prevenir errores 11.1

En general, los métodos `String` que toman índices inicial y final como argumentos incluyen al índice inicial y excluyen al índice final.

En la línea 25 se utiliza el método `getChars` de `String` para copiar los caracteres de una cadena en un arreglo de caracteres. El primer argumento es el índice inicial en la cadena, a partir del cual se van a copiar los caracteres. El segundo argumento es el índice que está una posición más adelante del último carácter que se va a copiar de la cadena. El tercer argumento es el arreglo de caracteres en el que se van a copiar los caracteres. El último argumento es el índice inicial en donde se van a colocar los caracteres copiados en el arreglo de caracteres. A continuación, el contenido del arreglo `char` se anexa, un carácter a la vez, a la cadena `salida` con la instrucción `for` en las líneas 28 y 29, para mostrarlo en pantalla.

11.3.3 Comparación entre cadenas

En el capítulo 7 hablamos sobre el ordenamiento y la búsqueda en los arreglos. Con frecuencia, la información que se va a ordenar o buscar consiste en cadenas que deben compararse para determinar el orden o para determinar si una cadena aparece en un arreglo (u otra colección). La clase `String` proporciona varios métodos para comparar cadenas, los cuales demostraremos en los siguientes dos ejemplos.

Para comprender lo que significa que una cadena sea "mayor que" o "menor que" otra cadena, considere el proceso de alfabetizar una serie de apellidos. Sin duda usted colocaría a "Jones" antes que "Smith", ya que en el alfabeto la "J" va antes que la "S". Pero el alfabeto es algo más que una lista de 26 letras; es un conjunto ordenado de caracteres. Cada letra ocupa una posición específica dentro del conjunto. "Z" es más que una letra del alfabeto; "Z" es específicamente la letra número veintiséis del alfabeto.

¿Cómo sabe la computadora que una letra va antes que otra? Todos los caracteres se representan en la computadora como códigos numéricos (vea el apéndice B). Cuando la computadora compara dos cadenas, en realidad compara los códigos numéricos de los caracteres en las cadenas.

http://librosysdiccionarios.net

En la figura 11.3 se demuestran los métodos `equals`, `equalsIgnoreCase`, `compareTo` y `regionMatches`, y se demuestra el uso del operador de igualdad == para comparar objetos `String`.

```
1  // Fig. 11.3: CompararString.java
2  // Los métodos equals, equalsIgnoreCase, compareTo y regionMatches de String.
3  import javax.swing.JOptionPane;
4
5  public class CompararString {
6
7     public static void main( String args[] )
8     {
9        String s1 = new String( "hola" );  // s1 es una copia de "hola"
10       String s2 = "adiós";
11       String s3 = "Feliz Cumpleaños";
12       String s4 = "feliz cumpleaños";
13
14       String salida = "s1 = " + s1 + "\ns2 = " + s2 + "\ns3 = " + s3 +
15          "\ns4 = " + s4 + "\n\n";
16
17       // probar igualdad
18       if ( s1.equals( "hola" ) )  // true
19          salida += "s1 es igual a \"hola\"\n";
20       else
21          salida += "s1 es distinta de \"hola\"\n";
22
23       // probar igualdad con ==
24       if ( s1 == "hola" )  // false; no son el mismo objeto
25          salida += "s1 es igual a \"hola\"\n";
26       else
27          salida += "s1 es distinta de \"hola\"\n";
28
29       // probar igualdad (ignorar mayúsculas)
30       if ( s3.equalsIgnoreCase( s4 ) )  // true
31          salida += "s3 es igual a s4\n";
32       else
33          salida += "s3 es distinta de s4\n";
34
35       // probar compareTo
36       salida += "\ns1.compareTo( s2 ) es " + s1.compareTo( s2 ) +
37          "\ns2.compareTo( s1 ) es " + s2.compareTo( s1 ) +
38          "\ns1.compareTo( s1 ) es " + s1.compareTo( s1 ) +
39          "\ns3.compareTo( s4 ) es " + s3.compareTo( s4 ) +
40          "\ns4.compareTo( s3 ) es " + s4.compareTo( s3 ) + "\n\n";
41
42       // probar regionMatches (susceptible a mayúsculas)
43       if ( s3.regionMatches( 0, s4, 0, 5 ) )
44          salida += "Los primeros 5 caracteres de s3 y s4 concuerdan\n";
45       else
46          salida += "Los primeros 5 caracteres de s3 y s4 no concuerdan\n";
47
48       // probar regionMatches (ignorar mayúsculas)
49       if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
50          salida += "Los primeros 5 caracteres de s3 y s4 concuerdan";
```

Figura 11.3 Comparaciones entre objetos `String`. (Parte 1 de 2.)

```
51          else
52             salida += "Los primeros 5 caracteres de s3 y s4 no concuerdan";
53
54          JOptionPane.showMessageDialog( null, salida,
55             "Comparaciones entre cadenas", JOptionPane.INFORMATION_MESSAGE );
56
57          System.exit( 0 );
58       }
59
60    } // fin de la clase CompararString
```

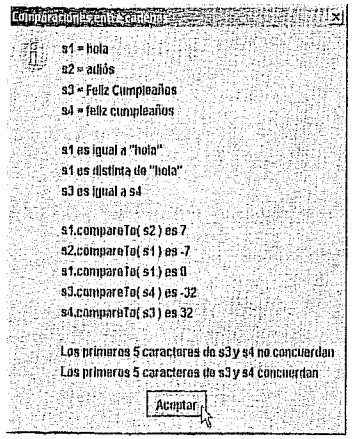


Figura 11.3 Comparaciones entre objetos `String`. (Parte 2 de 2.)

La condición en la instrucción `if...else` de la línea 18 utiliza el método `equals` para comparar la igualdad entre la cadena `s1` y la literal de cadena `"hola"`. El método `equals` (un método de la clase `Object`, redefinido en `String`) prueba la igualdad entre dos objetos (es decir, que las cadenas contenidas en los dos objetos sean idénticas). El método devuelve `true` si los objetos son iguales y `false` en caso contrario. La condición anterior es `true`, ya que la cadena `s1` fue inicializada con la literal de cadena `"hola"`. El método `equals` utiliza una *comparación lexicográfica* (se comparan los valores enteros Unicode que representan a cada uno de los caracteres en cada cadena). Por lo tanto, si la cadena `"hola"` se compara con la cadena `"HOLA"`, el resultado es `false` ya que la representación entera de una letra minúscula es distinta de la letra mayúscula correspondiente.

La condición en la instrucción `if...else` de la línea 24 utiliza el operador de igualdad `==` para comparar la igualdad entre la cadena `s1` y la literal de cadena `"hola"`. *El operador `==` tiene distinta funcionalidad cuando se utiliza para comparar referencias, que cuando se utiliza para comparar valores de tipos primitivos.* Cuando se comparan valores de tipos primitivos con `==`, el resultado es `true` si ambos valores son idénticos. Cuando se comparan referencias con `==`, el resultado es `true` si ambas referencias *se refieren al mismo objeto en memoria*. Para comparar la igualdad del contenido en sí (o información sobre el estado) de los objetos, deben invocarse métodos (como `equals`). La condición anterior se evalúa como `false` en la línea 24, ya que la referencia `s1` fue inicializada con la instrucción:

```
s1 = new String( "hola" );
```

con lo cual se crea un nuevo objeto `String` con una copia de la literal de cadena `"hola"`, y se asigna el nuevo objeto a la variable `s1`. Si `s1` se hubiera inicializado con la instrucción:

```
s1 = "hola";
```

http://librosysolucionarios.net

con lo cual se asignaría directamente la literal de cadena `"hola"` a la variable `s1`, la condición hubiera sido `true`. Recuerde que Java trata a todos los objetos de literal de cadena con el mismo contenido que un objeto `String` anónimo con muchas referencias. Por lo tanto, en las líneas 9, 18 y 24 se hace referencia al mismo objeto `String` anónimo `"hola"` en memoria.

Error común de programación 11.2



Al comparar referencias con `==` se pueden producir errores lógicos, ya que `==` compara las referencias para determinar si se refieren al mismo objeto, no si dos objetos tienen el mismo contenido. Si se comparan dos objetos idénticos (pero separados) con `==`, el resultado será `false`. Si va a comparar objetos para determinar si tienen el mismo contenido, utilice el método `equals`.

Si va a ordenar objetos `String`, puede utilizar el método `equalsIgnoreCase` para ver si son iguales. Este método no distingue el uso de mayúsculas y minúsculas al realizar la comparación. Por lo tanto, la cadena `"hola"` y la cadena `"HOLA"` se consideran iguales. La instrucción `if...else` de la línea 30 utiliza el método `equalsIgnoreCase` de `String` para comparar si la cadena `s3` (`Feliz Cumpleaños`) es igual a la cadena `s4` (`feliz cumpleaños`). El resultado de esta comparación es `true`, ya que la comparación ignora el uso de mayúsculas y minúsculas.

En las líneas 36 a 40 se utiliza el método `compareTo` de `String` para comparar cadenas. Por ejemplo, en la línea 36 se compara la cadena `s1` con la cadena `s2`. El método `compareTo` devuelve 0 si las cadenas son iguales, un número negativo si la cadena que invoca a `compareTo` es menor que la cadena que se pasa como argumento, y un número positivo si la cadena que invoca a `compareTo` es mayor que la cadena que se pasa como argumento. El método `compareTo` utiliza una comparación lexicográfica; es decir, compara los valores numéricos de los caracteres correspondientes en cada cadena.

La condición en la instrucción `if...else` de la línea 43 utiliza el método `regionMatches` de `String` para comparar si ciertas porciones de dos cadenas son iguales. El primer argumento es el índice inicial en la cadena que invoca al método. El segundo argumento es una cadena de comparación. El tercer argumento es el índice inicial en la cadena de comparación. El último argumento es el número de caracteres a comparar entre las dos cadenas. El método devuelve `true` solamente si el número especificado de caracteres es lexicográficamente igual.

Por último, la condición en la instrucción `if...else` de la línea 49 utiliza una segunda versión del método `regionMatches` de `String` para comparar si ciertas porciones de dos cadenas son iguales. Cuando el primer argumento es `true`, el método ignora (no distingue) el uso de mayúsculas y minúsculas en los caracteres a comparar. Los argumentos restantes son idénticos a los que se describieron para el método `regionMatches` de cuatro argumentos.

En el segundo ejemplo de esta sección (figura 11.4) demostraremos los métodos `startsWith` y `endsWith` de la clase `String`. El método `main` crea el arreglo `cadenas` que contiene las cadenas `"empezó"`, `"empezando"`, `"terminó"`, `"terminando"`. El resto del método `main` consiste en tres instrucciones `for` que prueban los elementos del arreglo para determinar si empiezan o terminan con un conjunto específico de caracteres.

```
1  // Fig. 11.4: InicioFinString.java
2  // String methods startsWith and endsWith.
3  import javax.swing.*;
4
5  public class InicioFinString {
6
7     public static void main( String args[] )
8     {
9        String cadenas[] = { "empezó", "empezando", "terminó", "terminando" };
10       String salida = "";
11
12       // probar el método startsWith
```

Figura 11.4 Los métodos `startsWith` y `endsWith` de la clase `String`. (Parte 1 de 2.)

```
13        for ( int cuenta = 0; cuenta < cadenas.length; cuenta++ )
14
15           if ( cadenas[ cuenta ].startsWith( "em" ) )
16              salida += "\"" + cadenas[ cuenta ] + "\" empieza con \"em\"\n";
17
18        salida += "\n";
19
20        // probar método startsWith, empezando desde la posición
21        // 2 de la cadena
22        for ( int cuenta = 0; cuenta < cadenas.length; cuenta++ )
23
24           if ( cadenas[ cuenta ].startsWith( "pez", 2 ) )
25              salida += "\"" + cadenas[ cuenta ] +
26                 "\" empieza con \"pez\" en la posición 2\n";
27
28        salida += "\n";
29
30        // probar el método endsWith
31        for ( int cuenta = 0; cuenta < cadenas.length; cuenta++ )
32
33           if ( cadenas[ cuenta ].endsWith( "do" ) )
34              salida += "\"" + cadenas[ cuenta ] + "\" termina con \"do\"\n";
35
36        JOptionPane.showMessageDialog( null, salida,
37           "Comparaciones con la clase String", JOptionPane.INFORMATION_MESSAGE );
38
39        System.exit( 0 );
40     }
41
42  } // fin de la clase InicioFinString
```

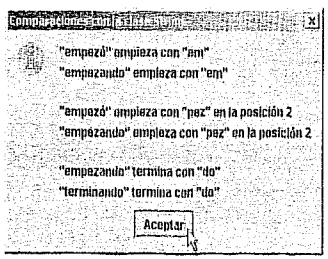


Figura 11.4 Los métodos **startsWith** y **endsWith** de la clase **String**. (Parte 2 de 2.)

La primera instrucción **for** (líneas 13 a 16) utiliza la versión del método **startsWith** que toma un argumento **String**. La condición en la instrucción **if** (línea 15) determina si la cadena en la posición **cuenta** del arreglo empieza con los caracteres **"em"**. De ser así, el método devuelve **true** y el programa anexa el valor de **cadenas[cuenta]** a **salida** para mostrar el resultado en pantalla. En caso contrario, el método devuelve **false** y no se anexa nada.

La segunda instrucción **for** (líneas 22 a 26) utiliza la versión del método **startsWith** que toma un **String** y un entero como argumentos. El argumento entero especifica el índice en el que debe empezar la comparación en la cadena. La condición en la instrucción **if** (línea 24) determina si la cadena en la posición **cuenta** del arreglo tiene los caracteres **"pez"**, empezando con el tercer carácter en cada cadena. De ser así, el método devuelve **true** y el programa anexa el valor de **cadenas[cuenta]** a **salida** para mostrar el resultado en pantalla.

http://librosysolucionarios.net

La tercera instrucción **for** (líneas 31 a 34) utiliza el método **endsWith** que toma un argumento **String**. La condición en la instrucción del **if** (línea 33) determina si la cadena en la posición **cuenta** del arreglo termina con los caracteres **"do"**. De ser así, el método devuelve **true** y el programa anexa el valor de **cadenas[cuenta]** a **salida** para mostrar el resultado en pantalla.

11.3.4 Localización de caracteres y subcadenas en las cadenas

A menudo es útil buscar un carácter o conjunto de caracteres en una cadena. Por ejemplo, si usted va a crear su propio procesador de palabras, tal vez quiera proporcionar una herramienta para buscar a través de los documentos. En la figura 11.5 se muestran las diversas versiones de los métodos ***indexOf*** y ***lastIndexOf***, que buscan un carácter o subcadena especificados en una cadena. Todas las búsquedas en este ejemplo se realizan en la cadena **letras** (inicializada con **"abcdefghijklmabcdefghijklm"**) que se encuentra en el método **main**.

```
1  // Fig. 11.5: MetodosIndexString.java
2  // String searching methods indexOf and lastIndexOf.
3  import javax.swing.*;
4
5  public class MetodosIndexString {
6
7     public static void main( String args[] )
8     {
9        String letras = "abcdefghijklmabcdefghijklm";
10
11       // probar indexOf para localizar un carácter en una cadena
12       String salida = "'c' se encuentra en el índice " + letras.indexOf( 'c' );
13
14       salida += "\n'a' se encuentra en el índice " + letras.indexOf( 'a', 1 );
15
16       salida += "\n'$' se encuentra en el índice " + letras.indexOf( '$' );
17
18       // probar lastIndexOf para buscar un carácter en una cadena
19       salida += "\n\nLa última 'c' se encuentra en el índice " +
20          letras.lastIndexOf( 'c' );
21
22       salida += "\nLa última 'a' se encuentra en el índice " +
23          letras.lastIndexOf( 'a', 25 );
24
25       salida += "\nEl último '$' se encuentra en el índice " +
26          letras.lastIndexOf( '$' );
27
28       // probar indexOf para localizar una subcadena en una cadena
29       salida += "\n\n\"def\" se encuentra en el índice " +
30          letras.indexOf( "def" );
31
32       salida += "\n\"def\" se encuentra en el índice " +
33          letras.indexOf( "def", 7 );
34
35       salida += "\n\"hola\" se encuentra en el índice " +
36          letras.indexOf( "hola" );
37
38       // probar lastIndexOf para encontrar una subcadena en una cadena
39       salida += "\n\nEl último \"def\" se encuentra en el índice " +
40          letras.lastIndexOf( "def" );
```

Figura 11.5 Los métodos de búsqueda de la clase **String**. (Parte 1 de 2.)

```
41
42          salida += "\nEl último \"def\" se encuentra en el índice " +
43             letras.lastIndexOf( "def", 25 );
44
45          salida += "\nEl último \"hola\" se encuentra en el índice " +
46             letras.lastIndexOf( "hola" );
47
48          JOptionPane.showMessageDialog( null, salida,
49             "Métodos de búsqueda de String", JOptionPane.INFORMATION_MESSAGE );
50
51          System.exit( 0 );
52       }
53
54    }  // fin de la clase MetodosIndexString
```

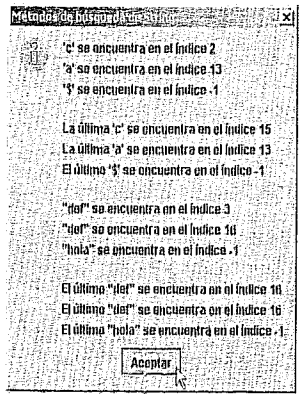


Figura 11.5 Los métodos de búsqueda de la clase `String`. (Parte 2 de 2.)

En las líneas 12 a 16 se utiliza el método `indexOf` para localizar la primera ocurrencia de un carácter en una cadena. Si `indexOf` encuentra el carácter, devuelve el índice de ese carácter en la cadena; en caso contrario `indexOf` devuelve –1. Hay dos versiones de `indexOf` que buscan caracteres en una cadena. La expresión en la línea 12 utiliza el método `indexOf` que toma un argumento entero, el cual es la representación entera de un carácter. Recuerde de la sección 11.2 que una literal de carácter entre comillas simples es de tipo `char` y especifica la representación entera de ese carácter en el conjunto de caracteres Unicode. La expresión en la línea 14 utiliza la segunda versión del método `indexOf`, que toma dos argumentos enteros: la representación entera de un carácter y el índice inicial en el que debe empezar la búsqueda en la cadena.

Las instrucciones de las líneas 19 a 26 utilizan el método `lastIndexOf` para localizar la última ocurrencia de un carácter en una cadena. El método `lastIndexOf` realiza la búsqueda desde el final de la cadena, hacia el inicio de la misma. Si el método `lastIndexOf` encuentra el carácter, devuelve el índice de ese carácter en la cadena; en caso contrario, devuelve –1. Hay dos versiones de `lastIndexOf` que buscan caracteres en una cadena. La expresión en la línea 20 utiliza la versión del método `lastIndexOf` que toma un argumento entero, el cual es la representación entera de un carácter. La expresión en la línea 23 utiliza la versión del método `lastIndexOf` que toma dos argumentos enteros: la representación entera de un carácter y el índice a partir del cual debe iniciarse la búsqueda inversa de ese carácter.

En las líneas 29 a 46 se demuestran las versiones de los métodos `indexOf` y `lastIndexOf` que toman, cada una de ellas, un objeto `String` como el primer argumento. Estas versiones de los métodos se ejecutan en forma idéntica a las descritas anteriormente, excepto que buscan secuencias de caracteres (o subcadenas) que se especifican mediante sus argumentos `String`. Si se encuentra la subcadena, estos métodos devuelven el índice en la cadena del primer carácter en la subcadena.

11.3.5 Cómo extraer subcadenas de las cadenas

La clase `String` proporciona dos métodos ***`substring`*** para permitir la creación de un nuevo objeto `String` al copiar parte de un objeto `String` existente. Cada método devuelve un nuevo objeto `String`. Ambos métodos se demuestran en la figura 11.6.

La expresión `letras.substring( 20 )` en la línea 13 utiliza el método `substring` que toma un argumento entero. Este argumento especifica el índice inicial en la cadena original `cadenas`, a partir del cual se van a copiar caracteres. La subcadena devuelta contiene una copia de los caracteres, desde el índice inicial hasta el final de la cadena. Si el índice especificado como argumento se encuentra fuera de los límites de la cadena, el programa genera una excepción `StringIndexOutOfBoundsException`.

La expresión `letras.substring( 3, 6 )` en la línea 16 utiliza el método `substring` que toma dos argumentos enteros. El primer argumento especifica el índice inicial a partir del que se van a copiar caracteres en la cadena original. El segundo argumento especifica el índice que está una posición más allá del último carácter a copiar (es decir, copiar hasta, pero sin incluir a, ese índice en la cadena). La subcadena devuelta contiene copias de los caracteres especificados de la cadena original. Si los argumentos están fuera de los límites de la cadena, el programa genera una excepción `StringIndexOutOfBoundsException`.

11.3.6 Concatenación de cadenas

El método `concat` (figura 11.7) de `String` concatena dos objetos `String` y devuelve un nuevo objeto `String`, el cual contiene los caracteres de ambos objetos `String` originales. Si el argumento `String` no

```
1  // Fig. 11.6: SubString.java
2  // Los métodos substring de la clase String.
3  import javax.swing.*;
4
5  public class SubString {
6
7     public static void main( String args[] )
8     {
9        String letras = "abcdefghijklmabcdefghijklm";
10
11       // probar métodos substring
12       String salida = "La subcadena desde el índice 20 hasta el final es " +
13          "\"" + letras.substring( 20 ) + "\"\n";
14
15       salida += "La subcadena desde el índice 3 hasta 6 es " +
16          "\"" + letras.substring( 3, 6 ) + "\"";
17
18       JOptionPane.showMessageDialog( null, salida,
19          "Métodos substring de la clase String",
20          JOptionPane.INFORMATION_MESSAGE );
21
22       System.exit( 0 );
23    }
24 } // fin de la clase SubString
```

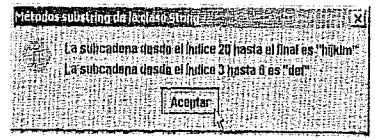


Figura 11.6 Los métodos `substring` de la clase `String`.

http://librosysolucionarios.net

contiene caracteres, se devuelve la cadena original. La expresión `s1.concat( s2 )` de la línea 14 forma una cadena, anexando los caracteres de la cadena `s2` a los caracteres de la cadena `s1`. Los objetos `String` originales a los que se refieren `s1` y `s2` no se modifican.

Tip de rendimiento 11.2

En los programas que realizan frecuentemente la concatenación de cadenas u otro tipo de modificaciones a las cadenas, es más eficiente implementar esas modificaciones con la clase `StringBuffer` *(que veremos en la sección 11.4).*

11.3.7 Métodos varios de `String`

La clase `String` proporciona varios métodos que devuelven copias modificadas de cadenas, o que devuelven arreglos de caracteres. Estos métodos se demuestran en la aplicación de la figura 11.8.

En la línea 17 se utiliza el método `replace` de `String` para devolver un nuevo objeto `String`, en el que cada ocurrencia del carácter `'l'` en la cadena `s1` se reemplaza con el carácter `'L'`. El método `replace` no modifica la cadena original. Si no hay ocurrencias del primer argumento en la cadena, el método `replace` devuelve la cadena original.

En la línea 20 se utiliza el método `toUpperCase` de `String` para generar un nuevo objeto `String` con letras mayúsculas, cuyas letras minúsculas correspondientes existen en `s1`. El método devuelve un nuevo objeto `String` que contiene la cadena convertida y deja la cadena original sin cambios. Si no hay caracteres qué convertir, el método `toUpperCase` devuelve la cadena original.

```
1  // Fig. 11.7: ConcatenacionString.java
2  // El método concat de la clase String.
3  import javax.swing.*;
4
5  public class ConcatenacionString {
6
7     public static void main( String args[] )
8     {
9        String s1 = new String( "Feliz " );
10       String s2 = new String( "Cumpleaños" );
11
12       String salida = "s1 = " + s1 + "\ns2 = " + s2;
13
14       salida += "\n\nResultado de of s1.concat( s2 ) = " + s1.concat( s2 );
15       salida += "\ns1 después de la concatenación = " + s1;
16
17       JOptionPane.showMessageDialog( null, salida,
18          "El método concat de String", JOptionPane.INFORMATION_MESSAGE );
19
20       System.exit( 0 );
21    }
22
23 } // fin de la clase ConcatenacionString
```

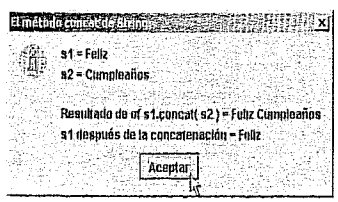


Figura 11.7 El método `concat` de la clase `String`.


```
1  // Fig. 11.8: VariosString2.java
2  // Los métodos replace, toLowerCase, toUpperCase, trim y
   //    toCharArray de String.
3  import javax.swing.*;
4
5  public class VariosString2 {
6
7     public static void main( String args[] )
8     {
9        String s1 = new String( "hola" );
10       String s2 = new String( "ADIÓS" );
11       String s3 = new String( "   espacios   " );
12
13       String salida = "s1 = " + s1 + "\ns2 = " + s2 + "\ns3 = " + s3;
14
15       // probar el método replace
16       salida += "\n\nReemplazar 'l' con 'L' en s1: " +
17          s1.replace( 'l', 'L' );
18
19       // probar toLowerCase y toUpperCase
20       salida += "\n\ns1.toUpperCase() = " + s1.toUpperCase() +
21          "\ns2.toLowerCase() = " + s2.toLowerCase();
22
23       // probar el método trim
24       salida += "\n\ns3 después de trim = \"" + s3.trim() + "\"";
25
26       // probar el método toCharArray
27       char arregloChar[] = s1.toCharArray();
28       salida += "\n\ns1 como un arreglo de caracteres = ";
29
30       for ( int cuenta = 0; cuenta < arregloChar.length; ++cuenta )
31          salida += arregloChar[ cuenta ];
32
33       JOptionPane.showMessageDialog( null, salida,
34          "Métodos adicionales de String", JOptionPane.INFORMATION_MESSAGE );
35
36       System.exit( 0 );
37    }
38
39 } // fin de la clase VariosString2
```

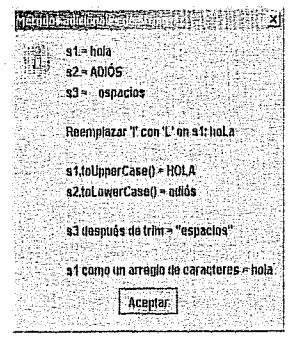


Figura 11.8 Los métodos `replace`, `toLowerCase`, `toUpperCase`, `trim`, `toString` y `toCharArray`.

http://librosysolucionarios.net

En la línea 21 se utiliza el método `toLowerCase` de `String` para devolver un nuevo objeto `String` con letras minúsculas, cuyas letras mayúsculas correspondientes existen en `s2`. La cadena original permanece sin cambios. Si no hay caracteres qué convertir en la cadena original, `toLowerCase` devuelve la cadena original.

En la línea 24 se utiliza el método `trim` de `String` para generar un nuevo objeto `String` que elimine todos los caracteres de espacio en blanco que aparecen al principio o al final en la cadena en la que opera `trim`. El método devuelve un nuevo objeto `String` que contiene a la cadena sin espacio en blanco a la izquierda o a la derecha. La cadena original permanece sin cambios.



Tip para prevenir errores 11.2

Invoque el método `trim` de `String` en cadenas introducidas por el usuario, para asegurar que cualquier espacio adicional que haya introducido el usuario accidentalmente al principio o al final de la cadena se elimine antes de procesarse.

En la línea 27 se crea un nuevo arreglo de caracteres, el cual contiene una copia de los caracteres en la cadena `s1` y asigna una referencia a la variable `arregloChar`.

11.3.8 El método `valueOf` de `String`

Como hemos visto, todo objeto en Java tiene un método `toString` que permite a un programa obtener la representación de cadena del objeto. Desafortunadamente, esta técnica no puede utilizarse con tipos primitivos, ya que éstos no tienen métodos. La clase `String` proporciona métodos `static` que toman un argumento de cualquier tipo y lo convierten en un objeto `String`. La clase `StringValueOf` (figura 11.9) demuestra el uso de los métodos `valueOf` de la clase `String`.

```
1  // Fig. 11.9: StringValueOf.java
2  // Los métodos valueOf de la clase String.
3  import javax.swing.*;
4
5  public class StringValueOf {
6
7     public static void main( String args[] )
8     {
9        char arregloChar[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
10       boolean valorBooleano = true;
11       char valorCharacter = 'Z';
12       int valorInteger = 7;
13       long valorLong = 10000000L;
14       float valorFloat = 2.5f; // el sufijo f indica que 2.5 es un float
15       double valorDouble = 33.333;
16       Object refObject = "hola"; // asignar cadena a una referencia Object
17
18       String salida = "arreglo char = " + String.valueOf( arregloChar ) +
19          "\nparte de arreglo char = " + String.valueOf( arregloChar, 3, 3 ) +
20          "\nboolean = " + String.valueOf( valorBooleano ) +
21          "\nchar = " + String.valueOf( valorCharacter ) +
22          "\nint = " + String.valueOf( valorInteger ) +
23          "\nlong = " + String.valueOf( valorLong ) +
24          "\nfloat = " + String.valueOf( valorFloat ) +
25          "\ndouble = " + String.valueOf( valorDouble ) +
26          "\nObject = " + String.valueOf( refObject );
27
28       JOptionPane.showMessageDialog( null, salida,
29          "Métodos valueOf de String", JOptionPane.INFORMATION_MESSAGE );
```

Figura 11.9 Los métodos `valueOf` de la clase `String`. (Parte 1 de 2.)

http://librosysolucionarios.net


```
30
31       System.exit( 0 );
32    }
33
34 } // fin de la clase StringValueOf
```

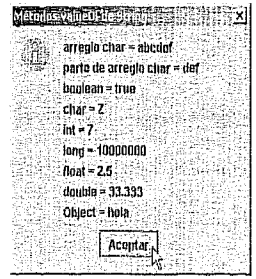


Figura 11.9 Los métodos `valueOf` de la clase `String`. (Parte 2 de 2.)

La expresión `String.valueOf( arregloChar )` en la línea 18 utiliza el arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. La expresión `String.valueOf( arregloChar, 3, 3 )` de la línea 19 utiliza una porción del arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. El segundo argumento especifica el índice inicial a partir del cual se van a utilizar caracteres. El tercer argumento especifica el número de caracteres a usar.

Existen otras siete versiones del método `valueOf`, las cuales toman argumentos de tipo `boolean`, `char`, `int`, `long`, `float`, `double` y `Object`, respectivamente. Estas versiones se demuestran en las líneas 20 a 26. Observe que la versión de `valueOf` que toma un `Object` como argumento puede hacerlo debido a que todos los objetos `Object` pueden convertirse en objetos `String` mediante el método `toString`.

(Nota: En las líneas 13 y 14 se utilizan los valores literales `10000000L` y `2.5f` como valores iniciales de la variable `valorLong` tipo `long` y de la variable `valorFloat` tipo `float`, respectivamente. De manera predeterminada, Java trata a las literales enteras como tipo `int` y a las literales de punto flotante como tipo `double`. Si se anexa la letra `L` a la literal `10000000` y se anexa la letra `f` a la literal `2.5`, se indica al compilador que `10000000` debe tratarse como `long` y que `2.5` debe tratarse como `float`.)

11.4 La clase `StringBuffer`

La clase `String` proporciona muchas herramientas para procesar cadenas. Sin embargo, una vez creado un objeto `String`, su contenido no puede cambiar. Ahora veremos las características de la clase `StringBuffer` para crear y manipular información en cadenas dinámicas (es decir, cadenas modificables). Todo objeto `StringBuffer` es capaz de almacenar un número de caracteres especificados según su capacidad. Si se excede la capacidad de un `StringBuffer`, se expande automáticamente para dar cabida a los caracteres adicionales. Como veremos, la clase `StringBuffer` también se utiliza para implementar los operadores + y += para concatenar cadenas.



Tip de rendimiento 11.3

Los objetos `String` son cadenas constantes, mientras que los objetos `StringBuffer` son objetos modificables. Java marca la diferencia entre las cadenas constantes y las cadenas modificables por cuestión de optimización; en especial, Java puede realizar ciertas optimizaciones en relación con los objetos `String` (como compartir un objeto `String` entre varias referencias), ya que sabe que estos objetos no cambiarán.



Tip de rendimiento 11.4

Cuando se tiene la opción entre utilizar un objeto `String` o un objeto `StringBuffer` para representar una cadena, siempre se debe utilizar un objeto `String` si la cadena no va a cambiar; esto mejora el rendimiento.

11.4.1 Constructores de `StringBuffer`

La clase **`StringBuffer`** proporciona tres constructores (los cuales se demuestran en la figura 11.10). En la línea 9 se utiliza el constructor predeterminado de **`StringBuffer`** para crear un objeto **`StringBuffer`** que no contiene caracteres, con una capacidad inicial de 16 caracteres (el valor predeterminado para un objeto **`StringBuffer`**). En la línea 10 se utiliza el constructor de **`StringBuffer`** que toma un argumento entero para crear un objeto **`StringBuffer`** que no contiene caracteres, con la capacidad inicial especificada por el argumento entero (es decir, **`10`**). En la línea 11 se utiliza el constructor de **`StringBuffer`** que toma un argumento **`String`** (en este caso, una literal de cadena) para crear un objeto **`StringBuffer`** que contiene los caracteres del argumento **`String`**. La capacidad inicial es el número de caracteres en el argumento **`String`**, más 16.

La instrucción en las líneas 13 a 15 utiliza el método **`toString`** de **`StringBuffer`** para convertir los objetos **`StringBuffer`** en objetos **`String`** que pueden mostrarse con **`drawString`**. Observe el uso del operador **`+`** para concatenar cadenas para mostrarlas en pantalla. En la sección 11.4.4 veremos cómo Java utiliza objetos **`StringBuffer`** para implementar los operadores **`+`** y **`+=`** para concatenar cadenas.

11.4.2 Los métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuffer`

La clase **`StringBuffer`** proporciona los métodos **`length`** y **`capacity`** para devolver el número de caracteres en un objeto **`StringBuffer`** y el número de caracteres que puede almacenarse en un objeto **`StringBuffer`** sin necesidad de asignar más memoria, respectivamente. El método **`ensureCapacity`** permite al programador garantizar que un **`StringBuffer`** tenga, cuando menos, la capacidad especificada. El método

```
1  // Fig. 11.10: ConstructoresStringBuffer.java
2  // Los constructores de la clase StringBuffer.
3  import javax.swing.*;
4
5  public class ConstructoresStringBuffer {
6
7     public static void main( String args[] )
8     {
9        StringBuffer buffer1 = new StringBuffer();
10       StringBuffer buffer2 = new StringBuffer( 10 );
11       StringBuffer buffer3 = new StringBuffer( "hola" );
12
13       String salida = "buffer1 = \"" + buffer1.toString() + "\"" +
14          "\nbuffer2 = \"" + buffer2.toString() + "\"" +
15          "\nbuffer3 = \"" + buffer3.toString() + "\"";
16
17       JOptionPane.showMessageDialog( null, salida,
18          "Constructores de StringBuffer", JOptionPane.INFORMATION_MESSAGE );
19
20       System.exit( 0 );
21    }
22
23 } // fin de la clase ConstructoresStringBuffer
```

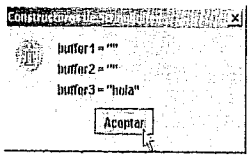


Figura 11.10 Los constructores de la clase **`StringBuffer`**.

`setLength` permite al programador incrementar o decrementar la longitud de un objeto **`StringBuffer`**. El programa de la figura 11.11 demuestra el uso de estos métodos.

El programa contiene un objeto **`StringBuffer`** llamado **`buffer`**. En la línea 9 se utiliza el constructor de **`StringBuffer`** que toma un argumento **`String`** para inicializar el objeto **`StringBuffer`** con la cadena **`"Hola, ¿cómo estás?"`**. En las líneas 11 y 12 se asigna a **`salida`** el contenido, la longitud y la capacidad del objeto **`StringBuffer`**. Observe en la ventana de salida que la capacidad del objeto **`StringBuffer`** es inicialmente de **`35`**. Recuerde de la sección 11.4.1 que el constructor de **`StringBuffer`** que toma un argumento **`String`** inicializa la capacidad con la longitud de la cadena que se le pasa como argumento, más 16.

En la línea 14 se utiliza el método **`ensureCapacity`** para expandir la capacidad del objeto **`StringBuffer`** a un mínimo de 75 caracteres. En realidad, si la capacidad original es menor que el argumento, el método asegura una capacidad que sea el valor mayor entre el número especificado como argumento, o el doble de la capacidad original más 2. Si la capacidad actual del objeto **`StringBuffer`** es más que la capacidad especificada, la capacidad del objeto **`StringBuffer`** permanece sin cambios.

```
1  // Fig. 11.11: StringBufferCapLen.java
2  // Los métodos length, setLength, capacity y ensureCapacity de StringBuffer.
3  import javax.swing.*;
4
5  public class StringBufferCapLen {
6
7     public static void main( String args[] )
8     {
9        StringBuffer buffer = new StringBuffer( "Hola, ¿cómo estás?" );
10
11       String salida = "buffer = " + buffer.toString() + "\nlongitud = " +
12          buffer.length() + "\ncapacidad = " + buffer.capacity();
13
14       buffer.ensureCapacity( 75 );
15       salida += "\n\nNueva capacidad = " + buffer.capacity();
16
17       buffer.setLength( 10 );
18       salida += "\n\nNueva longitud = " + buffer.length() +
19          "\nbuf = " + buffer.toString();
20
21       JOptionPane.showMessageDialog( null, salida,
22          "Los métodos length y capacity de StringBuffer",
23          JOptionPane.INFORMATION_MESSAGE );
24
25       System.exit( 0 );
26    }
27
28 } // fin de la clase StringBufferCapLen
```

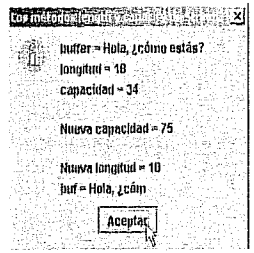


Figura 11.11 Los métodos **`length`** y **`capacity`** de **`StringBuffer`**.

http://librosysolucionarios.net

En la línea 17 se utiliza el método **setLength** para establecer la longitud del objeto **StringBuffer** en 10. Si la longitud especificada es menor que el número actual de caracteres en el objeto **StringBuffer**, el búfer se trunca a la longitud especificada (es decir, los caracteres en el objeto **StringBuffer** después de la longitud especificada se descartan). Si la longitud especificada es mayor que el número de caracteres actualmente en el objeto **StringBuffer**, se anexan caracteres nulos (caracteres con la representación numérica de 0) al objeto **StringBuffer** hasta que el número total de caracteres en el objeto **StringBuffer** sea igual a la longitud especificada.

11.4.3 Los métodos charAt, setCharAt, getChars y reverse de StringBuffer

La clase **StringBuffer** proporciona los métodos **charAt**, **setCharAt**, **getChars** y **reverse** para manipular los caracteres en un objeto **StringBuffer**. Cada uno de estos métodos se demuestra en la figura 11.12. El método **charAt** toma un argumento entero y devuelve el carácter que se encuentre en el objeto **StringBuffer** en ese índice. El método **setCharAt** toma un entero y un carácter como argumentos y asigna al carácter en la posición especificada en el objeto **StringBuffer** el carácter que recibe como argumento. El índice especificado en los métodos **charAt** y **setCharAt** debe ser mayor o igual que 0, y menor que la longitud del objeto **StringBuffer**; en caso contrario se produce una excepción **StringIndexOutOfBoundsException**.



Error común de programación 11.3

*Al tratar de acceder a un carácter que se encuentra fuera de los límites de un objeto **StringBuffer** (es decir, con un índice menor que 0, o mayor o igual que la longitud del objeto **StringBuffer**) se produce una excepción **StringIndexOutOfBoundsException**.*

El método **getChars** (línea 16) copia caracteres de un objeto **StringBuffer** en el arreglo de caracteres que recibe como argumento. Este método toma cuatro argumentos: el índice inicial a partir del cual deben copiarse los caracteres en el objeto **StringBuffer**, el índice que está una posición más allá del último carácter a copiar del objeto **StringBuffer**, el arreglo de caracteres en el que se van a copiar los caracteres y la ubicación inicial en el arreglo de caracteres en el que debe colocarse el primer carácter. El método **reverse** invierte el contenido del objeto **StringBuffer**.

11.4.4 Los métodos append de StringBuffer

La clase **StringBuffer** proporciona 11 métodos **append** sobrecargados para permitir que se agreguen varios tipos al final de un objeto **StringBuffer**. Se proporcionan versiones para cada uno de los tipos primitivos y para arreglos de caracteres, objetos **String**, **Object** y **StringBuffer**. (Recuerde que el método **toString** produce una representación de cadena de cualquier objeto **Object**.) Cada uno de los métodos toma su argumento, lo convierte en una cadena y lo anexa al objeto **StringBuffer**. Los métodos **append** se demuestran en la figura 11.13.

```
1  // Fig. 11.12: StringBufferChars.java
2  // Los métodos charAt, setCharAt, getChars y reverse de StringBuffer.
3  import javax.swing.*;
4
5  public class StringBufferChars {
6
7     public static void main( String args[] )
8     {
9        StringBuffer buffer = new StringBuffer( "hola a todos" );
10
11       String salida = "buffer = " + buffer.toString() +
12          "\nCaracter en 0: " + buffer.charAt( 0 ) +
13          "\nCaracter en 3: " + buffer.charAt( 3 );
```

Figura 11.12 Los métodos de la clase **StringBuffer** para manipular caracteres. (Parte 1 de 2.)


```
14
15       char arregloChar[] = new char[ buffer.length() ];
16       buffer.getChars( 0, buffer.length(), arregloChar, 0 );
17       salida += "\n\nLos caracteres son: ";
18
19       for ( int cuenta = 0; cuenta < arregloChar.length; ++cuenta )
20          salida += arregloChar[ cuenta ];
21
22       buffer.setCharAt( 0, 'H' );
23       buffer.setCharAt( 7, 'T' );
24       salida += "\n\nbuf = " + buffer.toString();
25
26       buffer.reverse();
27       salida += "\n\nbuf = " + buffer.toString();
28
29       JOptionPane.showMessageDialog( null, salida,
30          "Los métodos Chars de StringBuffer",
31          JOptionPane.INFORMATION_MESSAGE );
32
33       System.exit( 0 );
34    }
35
36 } // fin de la clase StringBufferChars
```

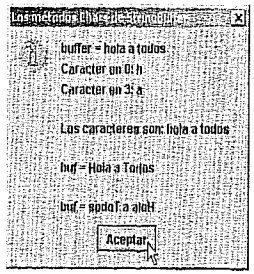


Figura 11.12 Los métodos de la clase **StringBuffer** para manipular caracteres. (Parte 2 de 2.)

```
1  // Fig. 11.13: StringBufferAppend.java
2  // StringBuffer append methods.
3  import javax.swing.*;
4
5  public class StringBufferAppend {
6
7     public static void main( String args[] )
8     {
9        Object refObject = "hola";
10       String cadena = "adiós";
11       char arregloChar[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
12       boolean valorBoolean = true;
13       char valorCharacter = 'Z';
14       int valorInteger = 7;
15       long valorLong = 10000000;
```

Figura 11.13 Los métodos **append** de la clase **StringBuffer**. (Parte 1 de 2.)

http://librosysolucionarios.net

```
16        float valorFloat = 2.5f; // el sufijo f indica que 2.5 es un float
17        double valorDouble = 33.333;
18        StringBuffer ultimoBuffer = new StringBuffer( "último StringBuffer" );
19        StringBuffer buffer = new StringBuffer();
20
21        buffer.append( refObject );
22        buffer.append( " " );  // cada uno de estos contiene dos espacios
23        buffer.append( cadena );
24        buffer.append( " " );
25        buffer.append( arregloChar );
26        buffer.append( " " );
27        buffer.append( arregloChar, 0, 3 );
28        buffer.append( " " );
29        buffer.append( valorBoolean );
30        buffer.append( " " );
31        buffer.append( valorCharacter );
32        buffer.append( " " );
33        buffer.append( valorInteger );
34        buffer.append( " " );
35        buffer.append( valorLong );
36        buffer.append( " " );
37        buffer.append( valorFloat );
38        buffer.append( " " );
39        buffer.append( valorDouble );
40        buffer.append( " " );
41        buffer.append( ultimoBuffer );
42
43        JOptionPane.showMessageDialog( null,
44           "buffer = " + buffer.toString(), "Métodos append de StringBuffer",
45           JOptionPane.INFORMATION_MESSAGE );
46
47        System.exit( 0 );
48     }
49
50  } // fin de StringBufferAppend
```

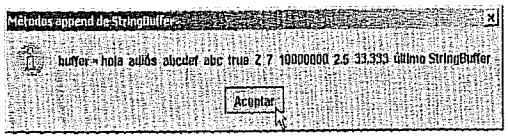


Figura 11.13 Los métodos **append** de la clase **StringBuffer**. (Parte 2 de 2.)

En realidad, los objetos **StringBuffer** y los métodos **append** son utilizados por el compilador para implementar los operadores + y += para concatenar objetos **String**. Por ejemplo, suponga que se realizan las siguientes declaraciones:

```
String cadena1 = "hola";
String cadena2 = "BC"
int valor = 22;
```

la instrucción

```
String s = cadena1 + cadena2 + valor;
```

concatena a **"hola"**, **"BC"** y **22**. La concatenación se realiza de la siguiente manera:

```
new StringBuffer().append( "hola" ).append( "BC" ).append( 22 ).toString();
```

http://librosysolucionarios.net

Primero, Java crea un objeto **StringBuffer** vacío y después anexa a este objeto **StringBuffer** las cadenas **"hola"**, **"BC"** y el entero **22**. A continuación, el método **toString** de **StringBuffer** convierte el objeto **StringBuffer** en una representación de cadena, y el resultado se asigna al objeto **String s**. La instrucción

```
s += "!";
```

se ejecuta de la siguiente manera:

```
s = new StringBuffer().append( s ).append( "!" ).toString()
```

Primero, Java crea un objeto **StringBuffer** vacío y después anexa a ese objeto **StringBuffer** el contenido de **s**, seguido por **"!"**. A continuación, el método **toString** de **StringBuffer** convierte el objeto **StringBuffer** en una representación de cadena, y el resultado se asigna a **s**.

11.4.5 Los métodos insert y delete de StringBuffer

La clase **StringBuffer** proporciona diez métodos **insert** sobrecargados, para permitir que se inserten varios tipos en cualquier posición de un objeto **StringBuffer**. Se proporcionan versiones para cada uno de los tipos primitivos, y para arreglos de caracteres, objetos **String** y **Object**. Cada uno de los métodos toma su segundo argumento, lo convierte en una cadena y la inserta antes del índice especificado por el primer argumento. El índice especificado por el primer argumento debe ser mayor o igual que 0, y menor que la longitud del objeto **StringBuffer**; de no ser así, se produce una excepción **StringIndexOutOfBoundsException**. La clase **StringBuffer** también proporciona métodos ***delete*** y ***deleteCharAt*** para eliminar caracteres en cualquier posición de un objeto **StringBuffer**. El método **delete** toma dos argumentos: el índice inicial y el índice que se encuentra una posición más allá del último de los caracteres que se van a eliminar. Se eliminan todos los caracteres que empiezan en el índice inicial hasta, pero sin incluir al índice final. El método **deleteCharAt** toma un argumento: el índice del carácter a eliminar. El uso de índices inválidos hace que ambos métodos lancen una excepción **StringIndexOutOfBoundsException**. Los métodos **insert** y **delete** se demuestran en la figura 11.14.

```
1  // Fig. 11.14: StringBufferInsert.java
2  // Los métodos insert y delete de StringBuffer.
3  import javax.swing.*;
4
5  public class StringBufferInsert {
6
7     public static void main( String args[] )
8     {
9        Object refObject = "hola";
10       String cadena = "adiós";
11       char arregloChar[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
12       boolean valorBoolean = true;
13       char valorCharacter = 'K';
14       int valorInteger = 7;
15       long valorLong = 10000000;
16       float valorFloat = 2.5f;  // el sufijo f indica que 2.5 es un float
17       double valorDouble = 33.333;
18       StringBuffer buffer = new StringBuffer();
19
20       buffer.insert( 0, refObject );
21       buffer.insert( 0, " " );  // cada uno de estos contienen dos espacios
22       buffer.insert( 0, cadena );
23       buffer.insert( 0, " " );
```

Figura 11.14 Los métodos **insert** y **delete** de **StringBuffer**. (Parte 1 de 2.)

```
24          buffer.insert( 0, arregloChar );
25          buffer.insert( 0, " " );
26          buffer.insert( 0, arregloChar, 3, 3 );
27          buffer.insert( 0, " " );
28          buffer.insert( 0, valorBoolean );
29          buffer.insert( 0, " " );
30          buffer.insert( 0, valorCharacter );
31          buffer.insert( 0, " " );
32          buffer.insert( 0, valorInteger );
33          buffer.insert( 0, " " );
34          buffer.insert( 0, valorLong );
35          buffer.insert( 0, " " );
36          buffer.insert( 0, valorFloat );
37          buffer.insert( 0, " " );
38          buffer.insert( 0, valorDouble );
39
40          String salida = "buffer después de inserciones:\n" + buffer.toString();
41
42          buffer.deleteCharAt( 10 );   // eliminar 5 en 2.5
43          buffer.delete( 2, 6 );       // eliminar .333 in 33.333
44
45          salida += "\n\nbuffer después de eliminaciones:\n" + buffer.toString();
46
47          JOptionPane.showMessageDialog( null, salida,
48             "Métodos insert/delete de StringBuffer",
               JOptionPane.INFORMATION_MESSAGE );
49
50          System.exit( 0 );
51       }
52
53    } // fin de la clase StringBufferInsert
```

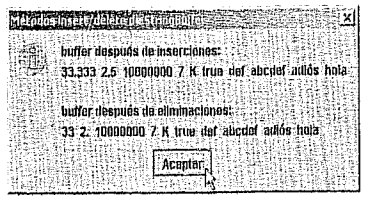


Figura 11.14 Los métodos `insert` y `delete` de `StringBuffer`. (Parte 2 de 2.)

11.5 La clase `Character`

En la sección 10.10 vimos que Java proporciona ocho clases de envoltura de tipos: `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` y `Long`, los cuales permiten que los valores de tipo primitivo sean tratados como objetos. En esta sección presentaremos la clase `Character`: la clase de envoltura de tipos para los caracteres.

La mayoría de los métodos de la clase `Character` son `static`, toman cuando menos un argumento tipo carácter y realizan una prueba o manipulación del carácter. Esta clase también contiene un constructor que recibe un argumento `char` para inicializar un objeto `Character`. En los siguientes tres ejemplos presentaremos la mayor parte de los métodos de la clase `Character`. Para obtener más información sobre esta clase (y las demás clases de envoltura de tipos), consulte el paquete `java.lang` en la documentación de JDK.

http://librosysolucionarios.net

En la figura 11.15 se demuestran algunos métodos `static` que prueban caracteres para determinar si son de un tipo de carácter específico y los métodos `static` que realizan conversiones de caracteres de minúscula a mayúscula y viceversa. Cada método se utiliza en el método `crearSalida` (líneas 51 a 65) de la clase `MetodosStaticChar`. Puede introducir cualquier carácter y aplicar estos métodos a ese carácter. Observe el uso de las clases internas anónimas para el manejo de eventos, como lo demostramos en el capítulo 9.

En la línea 53 se utiliza el método `isDefined` de `Character` para determinar si el carácter `c` está definido en el conjunto de caracteres Unicode. De ser así, el método devuelve `true`; en caso contrario, devuelve `false`.

```
1  // Fig. 11.15: MetodosStaticChar.java
2  // Métodos static de Character para probar caracteres y cambiar
      mayúsculas/minúsculas.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class MetodosStaticChar extends JFrame {
8     private char c;
9     private JLabel indicadorEtiqueta;
10    private JTextField entradaCampo;
11    private JTextArea areaSalida;
12
13    // el constructor crea la GUI
14    public MetodosStaticChar()
15    {
16       super( "Métodos estáticos de Character" );
17
18       Container contenedor = getContentPane();
19       contenedor.setLayout( new FlowLayout() );
20
21       indicadorEtiqueta = new JLabel( "Escriba un carácter y oprima Intro" );
22       contenedor.add( indicadorEtiqueta );
23       entradaCampo = new JTextField( 5 );
24
25       entradaCampo.addActionListener(
26
27          new ActionListener() {  // clase interna anónima
28
29             // manejar evento del campo de texto
30             public void actionPerformed( ActionEvent evento )
31             {
32                String s = evento.getActionCommand();
33                c = s.charAt( 0 );
34                crearSalida();
35             }
36
37          } // fin de la clase interna anónima
38
39       ); // fin de la llamada a addActionListener
40
41       contenedor.add( entradaCampo );
42       areaSalida = new JTextArea( 10, 20 );
43       contenedor.add( areaSalida );
```

Figura 11.15 Métodos estáticos de la clase `Character` para probar caracteres y convertir mayúsculas/minúsculas. (Parte 1 de 2.)

```
44
45          setSize( 300, 220 );  // establecer el tamaño de la ventana
46          setVisible( true);    // mostrar la ventana
47
48       } // fin del constructor
49
50       // mostrar información del carácter en areaSalida
51       private void crearSalida()
52       {
53          areaSalida.setText( "está definido: " + Character.isDefined( c ) +
54             "\nes dígito: " + Character.isDigit( c ) +
55             "\nes primer caracter en un identificador de Java: " +
56             Character.isJavaIdentifierStart( c ) +
57             "\nes parte de un identificador de Java: " +
58             Character.isJavaIdentifierPart( c ) +
59             "\nes letra: " + Character.isLetter( c ) +
60             "\nes letra o dígito: " + Character.isLetterOrDigit( c ) +
61             "\nes minúscula: " + Character.isLowerCase( c ) +
62             "\nes mayúscula: " + Character.isUpperCase( c ) +
63             "\na  mayúscula: " + Character.toUpperCase( c ) +
64             "\na  minúscula: " + Character.toLowerCase( c ) );
65       }
66
67       // crear objeto MetodosStaticChar para empezar la ejecución
68       public static void main( String args[] )
69       {
70          MetodosStaticChar aplicacion = new MetodosStaticChar();
71          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
72       }
73
74    } // fin de la clase MetodosStaticChar
```

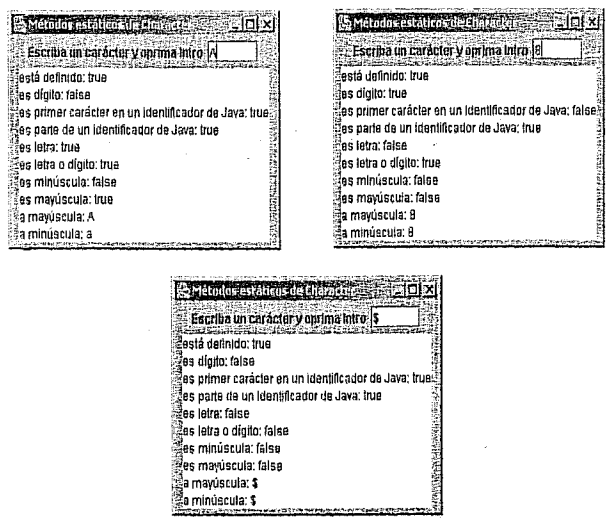


Figura 11.15 Métodos estáticos de la clase **Character** para probar caracteres y convertir mayúsculas/minúsculas. (Parte 2 de 2.)

http://librosysolucionarios.net

En la línea 54 se utiliza el método **isDigit** de **Character** para determinar si el carácter **c** es un dígito definido en Unicode. De ser así, el método devuelve **true**; en caso contrario devuelve **false**.

En la línea 56 se utiliza el método **isJavaIdentifierStart** de **Character** para determinar si **c** es un carácter que puede encontrarse en el primer carácter de un identificador en Java; es decir, una letra, un guión bajo (_) o un signo de dólares ($). De ser así, el método devuelve **true**; en caso contrario devuelve **false**. En la línea 58 se utiliza el método **isJavaIdentifierPart** de **Character** para determinar si el carácter **c** puede utilizarse en un identificador en Java; es decir, un dígito, una letra, un guión bajo (_) o un signo de dólares ($). De ser así, el método devuelve **true**; en caso contrario devuelve **false**.

En la línea 59 se utiliza el método **isLetter** de **Character** para determinar si el carácter **c** es una letra. Si es así, el método devuelve **true**; en caso contrario devuelve **false**. En la línea 60 se utiliza el método **isLetterOrDigit** de **Character** para determinar si el carácter **c** es una letra o un dígito. Si es así, el método devuelve **true**; en caso contrario devuelve **false**.

En la línea 61 se utiliza el método **isLowerCase** de **Character** para determinar si el carácter **c** es una letra minúscula. Si es así, el método devuelve **true**; en caso contrario devuelve **false**. En la línea 62 se utiliza el método **isUpperCase** de **Character** para determinar si el carácter **c** es una letra mayúscula. Si es así, el método devuelve **true**; en caso contrario devuelve **false**.

En la línea 63 se utiliza el método **toUpperCase** de **Character** para convertir el carácter **c** en su letra mayúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en mayúscula; en caso contrario, el método devuelve su argumento original. En la línea 64 se utiliza el método **toLowerCase** de **Character** para convertir el carácter **c** en su letra minúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en minúscula; en caso contrario, el método devuelve su argumento original.

En la figura 11.16 se muestran los métodos estáticos **digit** y **forDigit** de **Character**, los cuales convierten caracteres a dígitos y dígitos a caracteres, respectivamente, en distintos sistemas numéricos. Los sistemas numéricos comunes son el decimal (base 10), octal (base 8), hexadecimal (base 16) y binario (base 2). La base de un número se conoce también como su *raíz*. Para obtener más información sobre las conversiones entre sistemas numéricos, vea el apéndice C.

```
1  // Fig. 11.16: MetodosStaticChar2.java
2  // Métodos estáticos de Character para conversión de caracteres.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class MetodosStaticChar2 extends JFrame {
8     private char c;
9     private int digito, raiz;
10    private JLabel indicador1, indicador2;
11    private JTextField entrada, raizCampo;
12    private JButton enChar, enInt;
13
14    // el constructor crea la GUI
15    public MetodosStaticChar2()
16    {
17       super( "Métodos de conversión de Character" );
18
19       Container contenedor = getContentPane();
20       contenedor.setLayout( new FlowLayout() );
21
22       indicador1 = new JLabel( "Escriba un dígito o carácter " );
23       entrada = new JTextField( 5 );
24       contenedor.add( indicador1 );
```

Figura 11.16 Métodos estáticos de la clase **Character** para convertir cadenas. (Parte 1 de 3.)

```
25          contenedor.add( entrada );
26
27          indicador2 = new JLabel( "Escriba una raíz " );
28          raizCampo = new JTextField( 5 );
29          contenedor.add( indicador2 );
30          contenedor.add( raizCampo );
31
32          enChar = new JButton( "Convertir dígito en carácter" );
33          enChar.addActionListener(
34
35             new ActionListener() { // clase interna anónima
36
37                // manejar evento de objeto JButton enChar
38                public void actionPerformed( ActionEvent eventoAccion )
39                {
40                   digito = Integer.parseInt( entrada.getText() );
41                   raiz = Integer.parseInt( raizCampo.getText() );
42                   JOptionPane.showMessageDialog( null,
43                      "Convertir dígito en carácter: " +
44                      Character.forDigit( digito, raiz ) );
45                }
46
47             } // fin de clase interna anónima
48
49          ); // fin de la llamada a addActionListener
50
51          enInt = new JButton( "Convertir carácter en dígito" );
52          enInt.addActionListener(
53
54             new ActionListener() {  // clase interna anónima
55
56                // manejar evento de objeto JButton enInt
57                public void actionPerformed( ActionEvent eventoAccion )
58                {
59                   String s = entrada.getText();
60                   c = s.charAt( 0 );
61                   raiz = Integer.parseInt( raizCampo.getText() );
62                   JOptionPane.showMessageDialog( null,
63                      "Convertir carácter en dígito: " +
64                      Character.digit( c, raiz ) );
65                }
66
67             } // fin de la clase interna anónima
68
69          ); // fin de la llamada a addActionListener
70
71          contenedor.add( enChar );
72          contenedor.add( enInt );
73          setSize( 275, 150 );  // establecer el tamaño de la ventana
74          setVisible( true );   // mostrar la ventana
75       }
76
77       // crear objeto MetodosStaticChar2 para ejecutar la aplicación
78       public static void main( String args[] )
79       {
```

Figura 11.16 Métodos estáticos de la clase **Character** para convertir cadenas. (Parte 2 de 3.)


```
80          MetodosStaticChar2 aplicacion = new MetodosStaticChar2();
81          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
82       }
83
84    } // fin de la clase MetodosStaticChar2
```

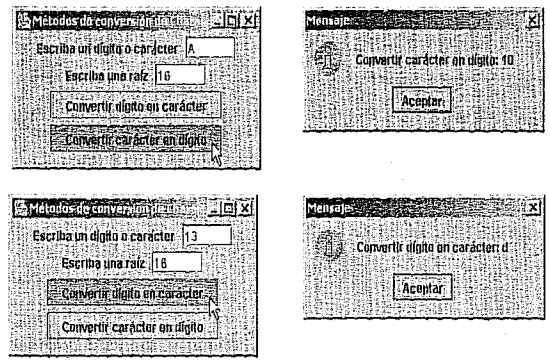


Figura 11.16 Métodos estáticos de la clase **Character** para convertir cadenas. (Parte 3 de 3.)

En la línea 44 se utiliza el método **forDigit** para convertir el entero **digito** en un carácter del sistema numérico especificado por el entero **raiz** (la base del número). Por ejemplo, el entero decimal **13** en base 16 (la **raiz**) tiene el valor de carácter **'d'**. Observe que las letras en minúsculas y mayúsculas son equivalentes en los sistemas numéricos.

En la línea 64 se utiliza el método **digit** para convertir el carácter **c** en un entero del sistema numérico especificado por el entero **raiz** (la base del número). Por ejemplo, el carácter **'A'** en base 16 (la **raiz**) tiene el valor entero de **10**.

El programa de la figura 11.17 demuestra el constructor y varios métodos no **static** de la clase **Character**: **charValue**, **toString** y **equals**.

En las líneas 9 a 10 se instancian dos objetos **Character** y se pasan literales de carácter al constructor para inicializar esos objetos.

En la línea 12 se utiliza el método **charValue** de **Character** para devolver el valor **char** almacenado en el objeto **Character** llamado **c1**. En la línea 13 se devuelve la representación de cadena del objeto **Character** llamado **c2**, utilizando el método **toString**.

La condición en la instrucción **if...else** de las líneas 15 a 18 utiliza el método **equals** para determinar si el objeto **c1** tiene el mismo contenido que el objeto **c2** (es decir, si los caracteres dentro de cada objeto son iguales).

11.6 La clase `StringTokenizer`

Cuando usted lee una oración, su mente la divide en *tokens* (palabras individuales y signos de puntuación, cada uno de los cuales transfiere a usted su significado). Los compiladores también llevan a cabo la descomposición de instrucciones en piezas individuales como palabras clave, identificadores, operadores y demás elementos de un lenguaje de programación. En esta sección estudiaremos la clase **StringTokenizer** de Java (del paquete **java.util**), la cual descompone una cadena en los tokens que la componen. Los tokens se separan unos de otros mediante *delimitadores*, que generalmente son caracteres de espacio en blanco como los espacios, tabuladores, nuevas líneas y retornos de carro. También pueden utilizarse otros caracteres como delimitadores para separar tokens. El programa de la figura 11.18 demuestra el uso de la clase **StringTokenizer**. La ventana para la clase **PruebaToken** muestra un objeto **JTextField** en donde el usuario escribe una oración para dividirla en tokens. La salida de este programa se muestra en un objeto **JTextArea**.

http://librosysolucionarios.net

```
1  // Fig. 11.17: OtrosMetodosChar.java
2  // Métodos no estáticos de Character.
3  import javax.swing.*;
4
5  public class OtrosMetodosChar {
6
7     public static void main( String args[] )
8     {
9        Character c1 = new Character( 'A' );
10       Character c2 = new Character( 'a' );
11
12       String salida = "c1 = " + c1.charValue() +
13          "\nc2 = " + c2.toString();
14
15       if ( c1.equals( c2 ) )
16          salida += "\n\nc1 y c2 son iguales";
17       else
18          salida += "\n\nc1 y c2 no son iguales";
19
20       JOptionPane.showMessageDialog( null, salida,
21          "Métodos no estáticos de Character",
22          JOptionPane.INFORMATION_MESSAGE );
23
24       System.exit( 0 );
25    }
26
27 } // fin de la clase OtrosMetodosChar
```

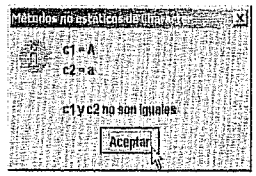


Figura 11.17 Métodos no estáticos de la clase **Character**.

```
1  // Fig. 11.18: PruebaToken.java
2  // La clase StringTokenizer.
3  import java.util.*;
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class PruebaToken extends JFrame {
9     private JLabel indicadorEtiqueta;
10    private JTextField entradaCampo;
11    private JTextArea areaSalida;
12
13    // configurar la GUI y manejo de eventos
14    public PruebaToken()
15    {
16       super( "Prueba de la clase StringTokenizer" );
```

http://librosysolucionarios.net

Figura 11.18 Objeto **StringTokenizer** utilizado para descomponer cadenas en tokens. (Parte 1 de 2.)


```
17
18       Container contenedor = getContentPane();
19       contenedor.setLayout( new FlowLayout() );
20
21       indicadorEtiqueta = new JLabel( "Escriba una oración y oprima Intro" );
22       contenedor.add( indicadorEtiqueta );
23
24       entradaCampo = new JTextField( 20 );
25       entradaCampo.addActionListener(
26
27          new ActionListener() {  // clase interna anónima
28
29             // manejar evento de campo de texto
30             public void actionPerformed( ActionEvent evento )
31             {
32                StringTokenizer tokens =
33                   new StringTokenizer( evento.getActionCommand() );
34
35                areaSalida.setText( "Número de elementos: " +
36                   tokens.countTokens() + "\nLos tokens son:\n" );
37
38                while ( tokens.hasMoreTokens() )
39                   areaSalida.append( tokens.nextToken() + "\n" );
40             }
41
42          } // fin de clase interna anónima
43
44       ); // fin de la llamada a addActionListener
45
46       contenedor.add( entradaCampo );
47
48       areaSalida = new JTextArea( 10, 20 );
49       areaSalida.setEditable( false );
50       contenedor.add( new JScrollPane( areaSalida ) );
51       setSize( 275, 240 );  // establecer el tamaño de la ventana
52       setVisible( true );   // mostrar la ventana
53    }
54
55    // ejecutar la aplicación
56    public static void main( String args[] )
57    {
58       PruebaToken aplicacion = new PruebaToken();
59       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
60    }
61
62 } // fin de la clase PruebaToken
```

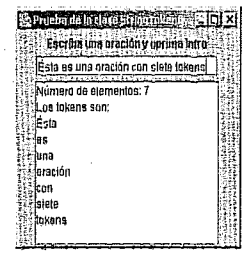


Figura 11.18 Objeto **StringTokenizer** utilizado para descomponer cadenas en tokens. (Parte 2 de 2.)

Cuando el usuario oprime *Intro* en el objeto `JTextField`, se invoca el método `actionPerformed` (líneas 30 a 40). En las líneas 32 y 33 se crea una instancia de la clase `StringTokenizer` utilizando la cadena en el campo de texto (`evento.getActionCommand()`) como un argumento. Este constructor de `StringTokenizer` toma un argumento de cadena y crea un objeto `StringTokenizer` para esa cadena, utilizando la cadena delimitadora predeterminada `" \t\n\r\f"` que consiste en un espacio, una nueva línea, un tabulador y un retorno de carro para la descomposición en tokens. Hay otros dos constructores para la clase `StringTokenizer`. En la versión que toma dos argumentos `String`, el segundo `String` es la cadena delimitadora. En la versión que toma tres argumentos, el segundo `String` es la cadena delimitadora y el tercer argumento (`boolean`) determina si los delimitadores también se devuelven como tokens (sólo si este argumento es `true`). Esto es útil si usted necesita saber cuáles son los delimitadores.

La instrucción en las líneas 38 y 39 utiliza el método `countTokens` de `StringTokenizer` para determinar el número de tokens en la cadena que se va a descomponer en tokens.

La condición en la instrucción `while` de las líneas 33 y 34 utiliza el método `hasMoreTokens` de `StringTokenizer` para determinar si hay más tokens en la cadena que va a descomponerse. De ser así, se invoca el método `append` para anexar en la cadena del objeto `JTextArea` llamado `areaSalida` el siguiente token. El siguiente token se obtiene mediante una llamada al método `nextToken` (línea 39) de `StringTokenizer`, el cual devuelve un objeto `String`. El token se imprime seguido de un carácter de nueva línea, de manera que los siguientes tokens aparezcan en líneas separadas.

Si desea cambiar la cadena delimitadora mientras descompone una cadena en tokens, puede hacerlo especificando una nueva cadena delimitadora en una llamada a `nextToken`, como se muestra a continuación:

```
tokens.nextToken( nuevaCadenaDelimitadora );
```

Esta característica no se demuestra en la figura 11.18.

11.7 Simulación para barajar y repartir cartas

En esta sección emplearemos la generación de números aleatorios para desarrollar un programa de simulación para barajar y repartir cartas. Este programa puede posteriormente utilizarse para implementar programas de juegos de cartas específicos.

Desarrollaremos la aplicación `PaqueteDeCartas` (figura 11.19), la cual crea un paquete de 52 cartas utilizando objetos `Carta`, y después permite al usuario repartir cada carta haciendo clic en un botón `"Repartir carta"`. Cada carta repartida se muestra en un objeto `JTextField`. El usuario puede también barajar el paquete en cualquier momento, haciendo clic en un botón `"Barajar cartas"`.

La clase `Carta` (líneas 129 a 146) contiene dos variables de instancia `String` (`cara` y `palo`) que se utilizan para guardar referencias al nombre de la cara y del palo para una `Carta` específica. El constructor de la clase (líneas 134 a 138) recibe dos objetos `String` que utiliza para inicializar a `cara` y `palo`. El método `toString` (líneas 141 a 144) se proporciona para crear un objeto `String` que consiste en la `cara` de la carta, la cadena `" de "` y el `palo` de la carta.

La clase `PaqueteDeCartas` (líneas 7 a 126) consiste en un arreglo llamado `paquete` de 52 referencias `Carta`, una variable entera `cartaActual` que representa la carta que se ha repartido más recientemente en el arreglo `paquete` (–1 si no se han repartido cartas todavía) y los componentes de la GUI utilizados para manipular el paquete de cartas. El método constructor de la aplicación crea la instancia del arreglo `paquete` (línea 23) y utiliza la instrucción `for` de las líneas 27 a 29 para llenar el arreglo `paquete` con objetos `Carta`. Cada `Carta` se instancia e inicializa con dos cadenas: una del arreglo `caras` (las cadenas del `"As"` hasta el `"Rey"`) y una del arreglo `palos` (`"Corazones"`, `"Diamantes"`, `"Tréboles"` y `"Espadas"`). El cálculo `cuenta % 13` siempre produce un valor del 0 al 12 (los trece índices del arreglo `caras`), y el cálculo `cuenta / 13` siempre produce un valor de 0 a 3 (los cuatro índices en el arreglo `palos`). Cuando se inicializa el arreglo `paquete`, éste contiene las cartas con las caras del as hasta el rey en orden para cada palo.

Cuando el usuario hace clic en el botón `Repartir carta`, el método `actionPerformed` de las líneas 41 a 53 invoca al método `repartirCarta` (declarado en las líneas 108 a 116) para obtener la siguiente carta en el arreglo. Si el `paquete` no está vacío, se devuelve una referencia a un objeto `Carta`; en caso contrario se devuelve `null`. Si la referencia no es `null`, en las líneas 46 a 47 se muestra la `Carta` en el campo de texto `pantallaCampo` y se muestra el número de la carta en la etiqueta `estadoEtiqueta`. Si la re-


```
1  // Fig. 11.19: PaqueteDeCartas.java
2  // Programa para barajar y repartir cartas.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PaqueteDeCartas extends JFrame {
8     private Carta paquete[];
9     private int cartaActual;
10    private JButton repartirBoton, barajarBoton;
11    private JTextField pantallaCampo;
12    private JLabel estadoEtiqueta;
13
14    // configurar paquete de cartas y GUI
15    public PaqueteDeCartas()
16    {
17       super( "Programa para repartir cartas" );
18
19       String caras[] = { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis",
20          "Siete", "Ocho", "Nueve", "Diez", "Joto", "Qüina", "Rey" };
21       String palos[] = { "Corazones", "Diamantes", "Tréboles", "Espadas" };
22
23       paquete = new Carta[ 52 ];
24       cartaActual = -1;
25
26       // llenar paquete con objetos Carta
27       for ( int cuenta = 0; cuenta < paquete.length; cuenta++ )
28          paquete[ cuenta ] = new Carta( caras[ cuenta % 13 ],
29             palos[ cuenta / 13 ] );
30
31       // configurar la GUI y manejo de eventos
32       Container contenedor = getContentPane();
33       contenedor.setLayout( new FlowLayout() );
34
35       repartirBoton = new JButton( "Repartir carta" );
36       repartirBoton.addActionListener(
37
38          new ActionListener() {  // clase interna anónima
39
40             // repartir una carta
41             public void actionPerformed( ActionEvent eventoAccion )
42             {
43                Carta repartida = repartirCarta();
44
45                if ( repartida != null ) {
46                   pantallaCampo.setText( repartida.toString() );
47                   estadoEtiqueta.setText( "Carta #: " + cartaActual );
48                }
49                else {
50                   pantallaCampo.setText( "NO HAY MÁS CARTAS QUE REPARTIR" );
51                   estadoEtiqueta.setText( "Barajee las cartas para continuar" );
52                }
53             }
54
55          } // fin de la clase interna anónima
```

Figura 11.19 Simulación para barajar y repartir cartas. (Parte 1 de 3.)

http://librosysolucionarios.net

```
56
57          ); // fin de la llamada a addActionListener
58
59          contenedor.add( repartirBoton );
60
61          barajarBoton = new JButton( "Barajar cartas" );
62          barajarBoton.addActionListener(
63
64             new ActionListener() {  // clase interna anónima
65
66                // barajar paquete
67                public void actionPerformed( ActionEvent eventoAccion )
68                {
69                   pantallaCampo.setText( "BARAJANDO ..." );
70                   barajar();
71                   pantallaCampo.setText( "SE BARAJÓ EL PAQUETE" );
72                }
73
74             } // fin de la clase interna anónima
75
76          ); // fin de la llamada a addActionListener
77
78          contenedor.add( barajarBoton );
79
80          pantallaCampo = new JTextField( 20 );
81          pantallaCampo.setEditable( false );
82          contenedor.add( pantallaCampo );
83
84          estadoEtiqueta = new JLabel();
85          contenedor.add( estadoEtiqueta );
86
87          setSize( 275, 120 );  // establecer tamaño de ventana
88          setVisible( true );   // mostrar ventana
89       }
90
91       // barajar paquete de cartas con algoritmo de una pasada
92       private void barajar()
93       {
94          cartaActual = -1;
95
96          // para cada carta, elegir otra carta aleatoria e intercambiarlas
97          for ( int primera = 0; primera < paquete.length; primera++ ) {
98             int segunda = ( int ) ( Math.random() * 52 );
99             Carta temp = paquete[ primera ];
100            paquete[ primera ] = paquete[ segunda ];
101            paquete[ segunda ] = temp;
102         }
103
104         repartirBoton.setEnabled( true );
105      }
106
107      // repartir una carta
108      private Carta repartirCarta()
109      {
110         if ( ++cartaActual < paquete.length )
```

Figura 11.19 Simulación para barajar y repartir cartas. (Parte 2 de 3.)


```
111            return paquete[ cartaActual ];
112         else {
113            repartirBoton.setEnabled( false );
114            return null;
115         }
116      }
117
118      // ejecutar aplicación
119      public static void main( String args[] )
120      {
121         PaqueteDeCartas aplicacion = new PaqueteDeCartas();
122
123         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
124      }
125
126   } // fin de la clase PaqueteDeCartas
127
128   // clase para representar una carta
129   class Carta {
130      private String cara;
131      private String palo;
132
133      // constructor para inicializar una carta
134      public Carta( String caraCarta, String paloCarta )
135      {
136         cara = caraCarta;
137         palo = paloCarta;
138      }
139
140      // devolver representación String de Carta
141      public String toString()
142      {
143         return cara + " de " + palo;
144      }
145
146   } // fin de la clase Carta
```



Figura 11.19 Simulación para barajar y repartir cartas. (Parte 3 de 3.)

ferencia devuelta por **repartirCarta** es **null**, se muestra la cadena **"NO HAY MÁS CARTAS QUE REPARTIR"** en el campo de texto, y la cadena **"Barajee las cartas para continuar"** en la etiqueta.

Cuando el usuario hace clic en el botón **Barajar cartas**, su método **actionPerformed** de las líneas 67 a 72 invoca al método **barajar** (declarado en las líneas 92 a 105) para barajar las cartas. El método itera a

http://librosysolucionarios.net

través de las 52 cartas (índices de arreglo 0 a 51). Para cada carta se elige al azar un número entre 0 y 51. A continuación, el objeto `Carta` actual y el objeto `Carta` elegido al azar se intercambian en el arreglo. Se realizan sólo un total de 52 intercambios en una sola pasada del arreglo completo, ¡y el arreglo de objetos `Carta` queda barajado! Cuando se termina de barajar el paquete, se muestra la cadena `"SE BARAJÓ EL PAQUETE"` en el objeto `JTextField`.

En las líneas 104 a 113 se utiliza el método `setEnabled` para activar y desactivar `repartirBoton`. El método `setEnabled` puede utilizarse en muchos componentes de la GUI. Cuando se llama con un argumento `false`, se deshabilita el componente de la GUI para el cual se llama, de manera que el usuario no pueda interactuar con él. Para reactivar el botón, llame a `setEnabled` con un argumento `true`.

11.8 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

Las expresiones regulares son una secuencia de caracteres y símbolos que definen un conjunto de cadenas. Son útiles para validar la entrada y asegurar que los datos estén en un formato específico. Por ejemplo, un código postal debe consistir de cinco dígitos, y un apellido debe empezar con letra mayúscula. Una aplicación de las expresiones regulares es facilitar la construcción de un compilador. A menudo se utiliza una expresión regular larga y compleja para validar la sintaxis de un programa. Si el código del programa no concuerda con la expresión regular, el compilador sabe que hay un error de sintaxis dentro del código. Las expresiones regulares son parte de la Nueva API de E/S de Java, la cual forma parte de la Plataforma Java 2, Standard Edition versión 1.4. Hablaremos sobre las características adicionales de la Nueva API de E/S de Java en los capítulos 17 y 18.

La clase `String` proporciona varios métodos para realizar operaciones con expresiones regulares, siendo la más simple la operación de concordancia. El método ***`matches`*** de la clase `String` recibe una cadena que especifica la expresión regular, y compara el contenido del objeto `String` que lo llama con la expresión regular. Este método devuelve un valor `boolean` indicando si hubo concordancia o no.

Una expresión regular consiste de caracteres literales y símbolos especiales. La tabla de la figura 11.20 especifica algunas *clases predefinidas de caracteres* que pueden usarse con las expresiones regulares. Una clase de carácter es una secuencia de escape que representa a un grupo de caracteres. Un *carácter de palabra* es cualquier carácter alfanumérico o guión bajo. Un carácter de *espacio en blanco* es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Un *dígito* es cualquier carácter numérico. Cada clase de carácter se compara con un solo carácter en la cadena que intentamos hacer concordar con la expresión regular.

Las expresiones regulares no están limitadas a esas clases predefinidas de caracteres. Las expresiones utilizan varios operadores y otras formas de notación para buscar patrones complejos. Analizaremos varias de estas técnicas en la aplicación de la figura 11.21, la cual valida la entrada del usuario mediante expresiones regulares.

Cuando un usuario hace clic en **Aceptar**, el programa comprueba que no haya campos vacíos (líneas 98 a 104). Si uno o más campos están vacíos, el programa informa al usuario que deben llenarse todos los campos para que el programa pueda validar la entrada (línea 106). Si no hay campos vacíos, se valida la entrada del usuario. En la línea 109 se valida el **primer nombre**. Para hacer que concuerde un conjunto de caracteres que no tiene una clase predefinida de carácter, utilice los corchetes (`[]`). Por ejemplo, el patrón `"[aeiou]"` puede utilizarse para concordar con una sola vocal. Los rangos de caracteres pueden representarse colocando un guión corto (-) entre dos caracteres. En el ejemplo, `"[A-Z]"` concuerda con una sola letra mayúscula. Si el primer carácter entre corchetes es `"^"`, la expresión acepta cualquier carácter distinto a los que se indiquen. Sin embargo, es importante observar que `"[^Z]"` no es lo mismo que `"[A-Y"]`, la cual concuerda con las letras mayúsculas A-Y; `"[^Z]"` concuerda con cualquier carácter que no esté en mayúscula (como 'a' o el

Carácter	Concuerda con	Carácter	Concuerda con
`\d`	cualquier dígito	`\D`	cualquier carácter que no sea dígito
`\w`	cualquier carácter de palabra	`\W`	cualquier carácter que no sea de palabra
`\s`	cualquier espacio en blanco	`\S`	cualquier carácter que no sea de espacio en blanco

Figura 11.20 Clases predefinidas de caracteres.

http://librosysolucionarios.net

carácter de nueva línea), además de concordar con las letras en mayúscula distintas de `Z`. Los rangos en las clases de caracteres se determinan mediante los valores enteros de las letras. En este ejemplo, `"[A-Za-z]"` concuerda con todas las letras mayúsculas y minúsculas. El rango `"[A-z]"` concuerda con todas las letras y también concuerda con los caracteres que tengan un valor entero entre la letra Z mayúscula y la letra a minúscula (como % y 6). Al igual que las clases predefinidas de caracteres, las clases de caracteres delimitadas entre corchetes concuerdan con un solo carácter en el objeto de búsqueda.

En la línea 109, el asterisco después de la segunda clase de carácter indica que puede concordar cualquier número de letras. En general, cuando aparece el operador de expresión regular `"*"` en una expresión regular, el programa intenta hacer que concuerden cero o más ocurrencias de la subexpresión que va inmediatamente de `"*"`. El operador `"+"` intenta hacer que concuerden una o más ocurrencias de la subexpresión que va inmediatamente después de `"+"`. Por lo tanto, `"A*"` y `"A+"` concordarán con `"A"`, pero sólo `"A*"` concordará con una cadena vacía.

```
1  // Fig. 11.21: ValidarFrame.java
2  // Validación de la información del usuario mediante expresiones regulares.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class ValidarFrame extends JFrame {
8     private JTextField telefonoCampoTexto, codigoPostalCampoTexto,
      estadoCampoTexto,
9        ciudadCampoTexto, direccionCampoTexto, nombreCampoTexto,
         apellidoCampoTexto;
10
11    public ValidarFrame()
12    {
13       super( "Validación" );
14
15       // crear los componentes de la GUI
16       JLabel telefonoEtiqueta = new JLabel( "Teléfono" );
17       JLabel codigoPostalEtiqueta = new JLabel( "C.P." );
18       JLabel estadoEtiqueta = new JLabel( "Estado" );
19       JLabel ciudadEtiqueta = new JLabel( "Ciudad" );
20       JLabel direccionEtiqueta = new JLabel( "Dirección" );
21       JLabel nombreEtiqueta = new JLabel( "Primer nombre" );
22       JLabel apellidoEtiqueta = new JLabel( "Apellido paterno" );
23
24       JButton aceptarBoton = new JButton( "Aceptar" );
25       aceptarBoton.addActionListener(
26
27          new ActionListener() { // clase interna
28
29             public void actionPerformed( ActionEvent evento ) {
30                validarFecha();
31             }
32
33          } // fin de la clase interna
34
35       ); // fin de la llamada a addActionListener
36
37       telefonoCampoTexto = new JTextField( 15 );
38       codigoPostalCampoTexto = new JTextField( 5 );
39       estadoCampoTexto = new JTextField( 2 );
```

Figura 11.21 Validación de la información del usuario mediante expresiones regulares. (Parte 1 de 4.)

```
40          ciudadCampoTexto = new JTextField( 12 );
41          direccionCampoTexto = new JTextField( 20 );
42          nombreCampoTexto = new JTextField( 20 );
43          apellidoCampoTexto = new JTextField( 20 );
44
45          JPanel primerNombre = new JPanel();
46          primerNombre.add( nombreEtiqueta );
47          primerNombre.add( nombreCampoTexto );
48
49          JPanel apellidoPaterno = new JPanel();
50          apellidoPaterno.add( apellidoEtiqueta );
51          apellidoPaterno.add( apellidoCampoTexto );
52
53          JPanel direccion1 = new JPanel();
54          direccion1.add( direccionEtiqueta );
55          direccion1.add( direccionCampoTexto );
56
57          JPanel direccion2 = new JPanel();
58          direccion2.add( ciudadEtiqueta );
59          direccion2.add( ciudadCampoTexto );
60          direccion2.add( estadoEtiqueta );
61          direccion2.add( estadoCampoTexto );
62          direccion2.add( codigoPostalEtiqueta );
63          direccion2.add( codigoPostalCampoTexto );
64
65          JPanel telefono = new JPanel();
66          telefono.add( telefonoEtiqueta );
67          telefono.add( telefonoCampoTexto );
68
69          JPanel aceptar = new JPanel();
70          aceptar.add( aceptarBoton );
71
72          // agregar los componentes a la aplicación
73          Container contenedor = getContentPane();
74          contenedor.setLayout( new GridLayout( 6, 1 ) );
75
76          contenedor.add( primerNombre );
77          contenedor.add( apellidoPaterno );
78          contenedor.add( direccion1 );
79          contenedor.add( direccion2 );
80          contenedor.add( telefono );
81          contenedor.add( aceptar );
82
83          setSize( 400, 225 );
84          setVisible( true );
85
86       } // fin del constructor de ValidarFrame
87
88       public static void main( String args[] )
89       {
90          ValidarFrame aplicacion = new ValidarFrame();
91          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
92       }
93
94       // maneja el evento de acción de aceptarBoton
```

Figura 11.21 Validación de la información del usuario mediante expresiones regulares. (Parte 2 de 4.)


```
95       private void validarFecha()
96       {
97          // asegurar que no haya cuadros de texto vacíos
98          if ( apellidoCampoTexto.getText().equals( "" ) ||
99               nombreCampoTexto.getText().equals( "" ) ||
100               direccionCampoTexto.getText().equals( "" ) ||
101               ciudadCampoTexto.getText().equals( "" ) ||
102               estadoCampoTexto.getText().equals( "" ) ||
103               codigoPostalCampoTexto.getText().equals( "" ) ||
104               telefonoCampoTexto.getText().equals( "" ) ) // fin de la condición
105
106             JOptionPane.showMessageDialog( this, "Por favor llene todos los
                campos" );
107
108          // si el formato de primer nombre es inválido, mostrar mensaje
109          else if ( !nombreCampoTexto.getText().matches( "[A-Z][a-zA-Z]*" ) )
110             JOptionPane.showMessageDialog( this, "Primer nombre inválido" );
111
112          // si el formato de apellido paterno es inválido, mostrar mensaje
113          else if ( !apellidoCampoTexto.getText().matches( "[A-Z][a-zA-Z]*" ) )
114             JOptionPane.showMessageDialog( this, "Apellido inválido" );
115
116          // si el formato de dirección es inválido, mostrar mensaje
117          else if ( !direccionCampoTexto.getText().matches(
118                "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" ) )
119             JOptionPane.showMessageDialog( this, "Dirección inválida" );
120
121          // si el formato de ciudad es inválido, mostrar mensaje
122          else if ( !ciudadCampoTexto.getText().matches(
123                "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" ) )
124             JOptionPane.showMessageDialog( this, "Ciudad inválida" );
125
126          // si el formato de estado es inválido, mostrar mensaje
127          else if ( !estadoCampoTexto.getText().matches(
128                "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" ) )
129             JOptionPane.showMessageDialog( this, "Estado inválido" );
130
131          // si el formato de código postal es inválido, mostrar mensaje
132          else if ( !codigoPostalCampoTexto.getText().matches( "\\d{5}" ) )
133             JOptionPane.showMessageDialog( this, "Código postal inválido" );
134
135          // si el formato de teléfono es inválido, mostrar mensaje
136          else if ( !telefonoCampoTexto.getText().matches(
137                "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" ) )
138             JOptionPane.showMessageDialog( this, "Número telefónico inválido" );
139
140          else // la información es válida, avisar al usuario
141             JOptionPane.showMessageDialog( this, "Gracias" );
142
143       } // fin del método validarFecha
144
145    } // fin de la clase ValidarFrame
```

http://librosysolucionarios.net

Figura 11.21 Validación de la información del usuario mediante expresiones regulares. (Parte 3 de 4.)

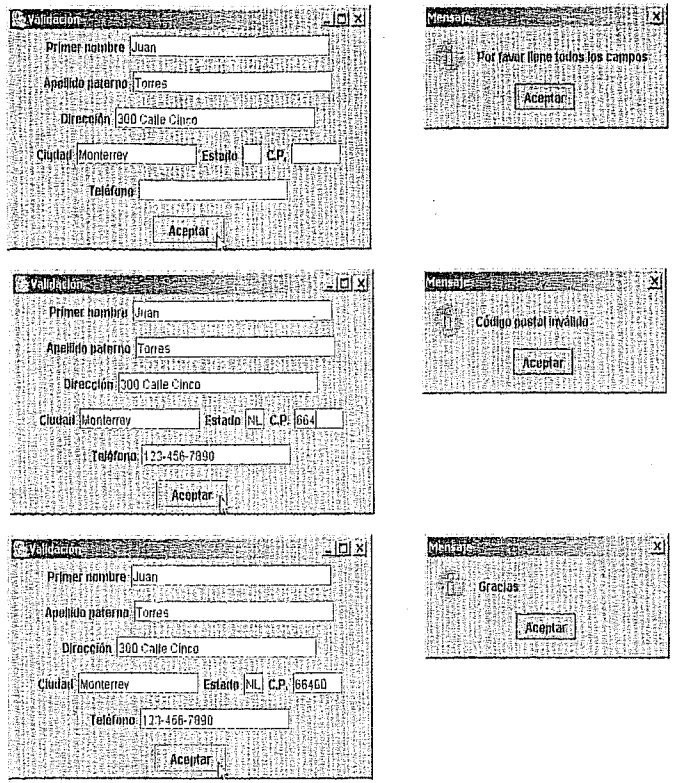


Figura 11.21 Validación de la información del usuario mediante expresiones regulares. (Parte 4 de 4.)

Si el método `matches` devuelve `true` (línea 109), el programa trata de validar el **apellido** (línea 113). La expresión regular para validar el apellido es idéntica a la que se utiliza para validar el primer nombre. De nuevo, esta expresión regular concordará con cualquier palabra que empiece con una letra mayúscula.

En las líneas 117 y 118 se valida la dirección. La primera clase de carácter concuerda con cualquier dígito una o más veces (`\\d+`). Observe que se utilizan dos caracteres \, ya que \ generalmente inicia una secuencia de escape en una cadena. Por lo tanto, `\\d` en una cadena de Java representa al patrón de expresión regular `\d`. Después concordamos uno o más caracteres de espacio en blanco (`\\s+`). El carácter "|" concuerda con la expresión a su izquierda o a su derecha. Por ejemplo, `Hola (Juan | Juana)` concuerda tanto con `Hola Juan` como con `Hola Juana`. Los paréntesis se utilizan para agrupar partes de la expresión regular. En este ejemplo, el lado izquierdo de | concuerda con una sola palabra y el lado derecho concuerda con dos palabras separadas por cierta cantidad de espacio en blanco. Por lo tanto, la dirección debe contener un número seguido de una o dos palabras. Así que, `"10 Broadway"` y `"10 Main Street"` son direcciones válidas en este ejemplo. Los campos **ciudad** (línea 122) y **estado** (línea 127) también concuerdan con cualquier palabra que tenga al menos un carácter o, de manera alternativa, con dos palabras cualesquiera con al menos un carácter, si éstas van separadas por un solo espacio. Esto significa que tanto `Waltham` como `West Newton` concordarían.

El asterisco (`*`) y el signo de suma (`+`) se conocen formalmente como *cuantificadores*. En la figura 11.22 se muestran varios cuantificadores. Ya hemos visto cómo funcionan el asterisco (`*`) y el signo de suma (`+`). Todos los cuantificadores afectan solamente a la subexpresión que va inmediatamente antes del cuantificador. El

cuantificador signo de interrogación (`?`) concuerda con cero o una ocurrencia de la expresión que cuantifica. Un conjunto de llaves que contienen un número (`{n}`) concuerda exactamente con `n` ocurrencias de la expresión que cuantifica. En la figura 11.21 demostramos este cuantificador para validar el código postal, en la línea 132. Si se incluye una coma después del número encerrado entre llaves, el cuantificador concordará al menos con `n` ocurrencias de la expresión cuantificada. El conjunto de llaves que contienen dos números (`{n,m}`) concuerda entre `n` y `m` ocurrencias de la expresión que califica. Los cuantificadores pueden aplicarse a patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Todos los cuantificadores son *avaros*. Esto significa que concordarán con todas las ocurrencias que puedan, siempre y cuando haya concordancia. No obstante, si alguno de estos cuantificadores va seguido por un signo de interrogación (`?`), el cuantificador se vuelve *reacio* (o, en algunas ocasiones, *flojo*). De esta forma, concordará con la menor cantidad de ocurrencias posibles, siempre y cuando haya concordancia.

El código postal (figura 11.21, línea 132) concuerda con un dígito cinco veces. Esta expresión regular utiliza la clase de carácter de dígito y un cuantificador con el dígito cinco entre llaves. El número telefónico (línea 136) concuerda con tres dígitos (el primero no puede ser cero) seguidos de un guión corto, seguido de tres dígitos más (de nuevo, el primero no puede ser cero), seguidos de cuatro dígitos más.

El método `matches` comprueba si toda una cadena completa se conforma a una expresión regular. Por ejemplo, suponga que queremos aceptar `"Smith"` como apellido, pero no `"9@Smith#"`. El método `matches` determina si toda la cadena completa concuerda con la expresión regular. Si sólo una subcadena concuerda con la expresión regular, el método `matches` devuelve `false`.

Reemplazo de subcadenas y división de cadenas

En ocasiones es conveniente reemplazar partes de una cadena, o dividir una cadena en varias piezas. Para este fin, la clase `String` proporciona los métodos `replaceAll`, `replaceFirst` y `split`. Estos métodos se demuestran en la figura 11.23.

Cuantificador	Concuerda con
`*`	Concuerda con cero o más ocurrencias del patrón.
`+`	Concuerda con una o más ocurrencias del patrón.
`?`	Concuerda con cero o una ocurrencia del patrón.
`{n}`	Concuerda con exactamente `n` ocurrencias.
`{n,}`	Concuerda con al menos `n` ocurrencias.
`{n,m}`	Concuerda con entre `n` y `m` (inclusive) ocurrencias.

Figura 11.22 Cuantificadores utilizados en expresiones regulares.

```
1  // Fig. 11.23: SustitucionRegex.java
2  // Uso de los métodos replaceFirst, replaceAll y split.
3  import javax.swing.*;
4
5  public class SustitucionRegex
6  {
7     public static void main( String args[] )
8     {
9        String primerString = "Este enunciado termina con 5 estrellas *****";
10       String segundoString = "1, 2, 3, 4, 5, 6, 7, 8";
11
12       String salida = "String original 1: " + primerString;
13
```

Figura 11.23 Los métodos `replaceFirst`, `replaceAll` y `split`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
14        // reemplazar '*' con '^'
15        primerString = primerString.replaceAll( "\\*", "^" );
16
17        salida += "\n^ reemplazó a *: " + primerString;
18
19        // reemplazar 'estrellas' con 'intercaladores'
20        primerString = primerString.replaceAll( "estrellas", "intercaladores" );
21
22        salida += "\n\"intercaladores\" reemplazan a \"estrellas\": " + primerString;
23
24        // reemplazar palabras con 'palabra'
25        salida += "\nCada palabra reemplazada por \"palabra\": " +
26           primerString.replaceAll( "\\w+", "palabra" );
27
28        salida += "\n\nString original 2: " + segundoString;
29
30        // reemplazar primeros tres dígitos con 'dígito'
31        for ( int i = 0; i < 3; i++ )
32           segundoString = segundoString.replaceFirst( "\\d", "dígito" );
33
34        salida += "\nPrimeros 3 dígitos reemplazados por \"dígito\" : " +
35           segundoString;
36        salida += "\nString dividido en las comas: [";
37
38        String[] resultados = segundoString.split( ",\\s*" ); // dividir en comas
39
40        for ( int i = 0; i < resultados.length; i++ )
41           salida += "\"" + resultados[ i ] + "\", "; // mostrar resultados
42
43        // eliminar la coma adicional y agregar un corchete
44        salida = salida.substring( 0, salida.length() - 2 ) + "]";
45
46        JOptionPane.showMessageDialog( null, salida );
47        System.exit( 0 );
48
49     } // fin del método main
50
51  } // fin de la clase SustitucionRegex
```

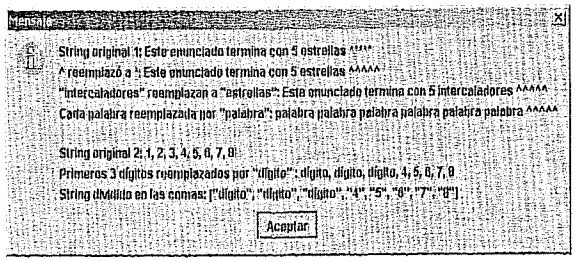


Figura 11.23 Los métodos `replaceFirst`, `replaceAll` y `split`. (Parte 2 de 2.)

El método `replaceAll` reemplaza el texto en una cadena con nuevo texto (el segundo argumento) en cualquier parte en donde la cadena original concuerde con una expresión regular (el primer argumento). En la línea 15 se reemplaza cada instancia de `"*"` en `primerString` con `"^"`. Observe que la expresión regular

(`"\\*"`) coloca dos barras diagonales inversas (\) antes del carácter `*`. Por lo general, `*` es un cuantificador que indica que una expresión regular debe concordar con cualquier número de ocurrencias del patrón que se coloca antes de este carácter. Sin embargo, en la línea 15 queremos encontrar todas las ocurrencias del carácter literal `*`; para ello, debemos escapar el carácter `*` con el carácter \. Al escapar un carácter especial de expresión regular con una \, indicamos al motor de concordancia de expresiones regulares que busque el carácter en sí, en vez de lo que representa en una expresión regular. Como la expresión está almacenada en una cadena de Java y \ es un carácter especial en las cadenas de Java, debemos incluir una \ adicional. Por lo tanto, la cadena de Java `"\\*"` representa el patrón de expresión regular `\*`, que concuerda con un solo carácter `'*'` en la cadena de búsqueda. En la línea 20, todas las coincidencias con la expresión regular `"estrellas"` en `primerString` se reemplazan con `"intercaladores"`.

El método `replaceFirst` (línea 32) reemplaza la primera ocurrencia de la concordancia de un patrón. Las cadenas de Java son inmutables, por lo cual el método `replaceFirst` devuelve una nueva cadena en la que se han reemplazado los caracteres apropiados. Esta línea toma la cadena original y la reemplaza con la cadena devuelta por `replaceFirst`. Al iterar tres veces, reemplazamos las primeras tres instancias de un dígito (`\d`) en `segundoString` con el texto `"dígito"`.

El método `split` divide una cadena en varias subcadenas. La cadena original se divide en cualquier posición que concuerde con una expresión regular especificada. El método `split` devuelve un arreglo de cadenas que contiene las subcadenas que resultan de cada concordancia con la expresión regular. En la línea 38 utilizamos el método `split` para descomponer en tokens una cadena de enteros separados por comas. El argumento es la expresión regular que localiza el delimitador. En este caso, utilizamos la expresión regular `",\\s*"` para separar las subcadenas siempre que haya una coma. Al concordar con cualquier carácter de espacio en blanco, eliminamos los espacios adicionales de las subcadenas resultantes. Observe que las comas y los espacios en blanco no se devuelven como parte de las subcadenas. De nuevo, observe que la cadena de Java `",\\s*"` representa la expresión regular `,\s*`.

Las clases `Pattern` y `Matcher`

Además de las herramientas para uso de expresiones regulares de la clase `String`, Java proporciona otras clases en el paquete `java.util.regex` que ayudan a los desarrolladores a manipular expresiones regulares. La clase `Pattern` representa una expresión regular. La clase `Matcher` contiene tanto un patrón de expresión regular como un objeto `CharSequence` en el que se va a buscar ese patrón.

`CharSequence` es una interfaz que permite el acceso de lectura a una secuencia de caracteres. Esta interfaz requiere que se declaren los métodos `charAt`, `length`, `subSequence` y `toString`. Tanto `String` como `StringBuffer` implementan la interfaz `CharSequence`, por lo que puede usarse una instancia de cualquiera de estas clases con la clase `Matcher`.



Error común de programación 11.4

Una expresión regular puede compararse con un objeto de cualquier clase que implemente a la interfaz `CharSequence`, pero la expresión regular debe ser un objeto `String`. Si se intenta crear una expresión regular como un objeto `StringBuffer` se produce un error.

Si se va a utilizar una expresión regular sólo una vez, puede usarse el método estático `matches` de la clase `Pattern`. Este método toma una cadena que especifica la expresión regular y un objeto `CharSequence` en el que se va a realizar la prueba de concordancia. Este método devuelve un valor `boolean`, el cual indica si el objeto de búsqueda (el segundo argumento) concuerda con la expresión regular.

Si se va a utilizar una expresión regular más de una vez, es más eficiente usar el método estático `compile` de la clase `Pattern`, para crear un objeto `Pattern` específico para esa expresión regular. Este método recibe una cadena que representa el patrón, y devuelve un nuevo objeto `Pattern`, el cual puede utilizarse para llamar al método `matcher`. Este método recibe un objeto `CharSequence` en el que se va a realizar la búsqueda, y devuelve un objeto `Matcher`.

La clase `Matcher` cuenta con el método `matches`, el cual realiza la misma tarea que el método `matches` de `Pattern`, pero no recibe argumentos; el patrón y el objeto de búsqueda están encapsulados en el objeto `Matcher`. La clase `Matcher` proporciona otros métodos, incluyendo `find`, `lookingAt`, `replaceFirst` y `replaceAll`.

http://librosysolucionarios.net

En la figura 11.24 presentamos un ejemplo sencillo en el que se utilizan expresiones regulares. Este programa compara las fechas de cumpleaños con una expresión regular. La expresión concuerda solamente con los cumpleaños que no ocurran en abril y que pertenezcan a personas cuyos nombres empiecen con "J".

En las líneas 13 y 14 se crea un objeto `Pattern` mediante la invocación al método estático `compile` de la clase `Pattern`. El carácter de punto `"."` en la expresión regular (línea 14) concuerda con cualquier carácter individual, excepto un carácter de nueva línea.

En la línea 22 se crea el objeto `Matcher` para la expresión regular compilada y la secuencia de concordancia (`cadena1`). En las líneas 24 y 25 se utiliza un ciclo `while` para iterar a través de la cadena. En la línea 24 se utiliza el método `find` de la clase `Matcher` para tratar de hacer que concuerde una pieza del objeto de búsqueda con el patrón de búsqueda. Cada una de las llamadas a este método empieza en el punto en el que terminó la última llamada, por lo que pueden encontrarse varias concordancias. El método `lookingAt` de la clase `Matcher` funciona de manera similar, sólo que siempre comienza desde el principio del objeto de búsqueda, y siempre encontrará la primera concordancia, si es que hay una.

```
1  // Fig. 11.24: ConcordanciasExpReg.java
2  // Demonstrating Classes Pattern and Matcher.
3  import java.util.regex.*;
4  import javax.swing.*;
5
6  class ConcordanciasExpReg
7  {
8     public static void main( String args[] )
9     {
10       String salida = "";
11
12       // crear expresión regular
13       Pattern expresion =
14          Pattern.compile( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
15
16       String cadena1 = "Jacinto nació el 05-12-75\n" +
17          "David nació el 11-04-68\n" +
18          "Miguel nació el 04-28-73\n" +
19          "Juan nació el 12-17-77";
20
21       // comparar expresión regular con la cadena e imprimir concordancias
22       Matcher matcher = expresion.matcher( cadena1 );
23
24       while ( matcher.find() )
25          salida += matcher.group() + "\n";
26
27       JOptionPane.showMessageDialog( null, salida );
28       System.exit( 0 );
29
30    } // fin de main
31
32 } // fin de la clase ConcordanciasExpReg
```

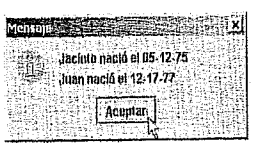


Figura 11.24 Expresiones regulares para comprobar fechas de nacimiento.

http://librosysolucionarios.net

Error común de programación 11.5

El método `matches` (de las clases `String`, `Pattern` o `Matcher`) devuelve `true` solamente si todo el objeto de búsqueda concuerda con la expresión regular. Los métodos `find` y `lookingAt` (de la clase `Matcher`) devuelven `true` si una parte del objeto de búsqueda concuerda con la expresión regular.

En la línea 25 se utiliza el método `group` de la clase `Matcher`, el cual devuelve la cadena del objeto de búsqueda que concuerda con el patrón de búsqueda. La cadena devuelta es la que haya concordado la última vez en una llamada a `find` o `lookingAt`. La salida en la figura 11.24 muestra las dos concordancias que se encontraron en `cadena1`.

Recursos en Web sobre expresiones regulares

En esta sección presentamos varias de las herramientas para expresiones regulares de Java. Los siguientes sitios Web proporcionan más información sobre las expresiones regulares.

```
developer.java.sun.com/developer/technicalArticles/releases/1.4regex
```

Este sitio incluye una descripción detallada de las herramientas para expresiones regulares del lenguaje Java.

```
java.sun.com/docs/books/tutorial/extra/regex/index.html
```

Este tutorial explica cómo utilizar la API de expresiones regulares que viene con Java.

```
java.sun.com/j2se/1.4.1/docs/api/java/util/regex/package-summary.html
```

Esta página es el panorama general del paquete `java.util.regex` mediante el uso de javadoc.

11.9 (Ejemplo práctico opcional) Acerca de los objetos: Manejo de eventos

Comúnmente, los objetos no realizan sus operaciones en forma espontánea. Por lo general, se invoca una operación específica cuando un objeto envía un *evento* (o mensaje) a un objeto receptor. En secciones anteriores mencionamos que los objetos interactúan enviando y recibiendo mensajes. Empezamos a modelar el comportamiento de nuestro sistema de elevador utilizando diagramas de estados y de actividad en la sección 5.11, y diagramas de colaboración en la sección 7.10. En esta sección veremos cómo interactúan los objetos del sistema de elevador.

Eventos

En la figura 7.18 presentamos un ejemplo de cómo una persona oprime un botón, enviando un mensaje `oprimirBoton` al botón (específicamente, el objeto `Persona` llamó al método `oprimirBoton` del objeto `Boton`). Este mensaje describe una acción que está ocurriendo en ese momento; en otras palabras, la `Persona` oprime un `Boton`. En general, la estructura de nombres de los mensajes es un verbo antes de un sustantivo (por ejemplo, el nombre del mensaje `oprimirBoton` consiste en el verbo "oprimir" seguido del sustantivo "botón".

Un evento es un mensaje que notifica a un objeto sobre una acción que ya ha ocurrido. Por ejemplo, en esta sección modificaremos nuestra simulación de manera que el `Elevador` envíe un evento `elevadorLlego` a la `Puerta` del `Elevador`, cuando éste llegue a un `Piso`. En la sección 7.10, el `Elevador` abre esta `Puerta` directamente, enviando un mensaje `abrirPuerta`. Al estar a la espera de escuchar un evento `elevadorLlego`, la `Puerta` puede determinar las acciones apropiadas a tomar cuando el `Elevador` haya llegado, tales como notificar a la `Persona` que la `Puerta` se ha abierto. Esto refuerza el principio del encapsulamiento en el DOO y modela el mundo real con más exactitud. En realidad, la puerta (y no el elevador) es la que "notifica" a la persona cuando se abre una puerta.

Observe que la estructura para nombrar eventos está invertida, en comparación con la estructura para nombrar el primer tipo de mensajes. Por convención, el nombre de un evento consiste en un sustantivo antes que el verbo. Por ejemplo, el nombre del evento `elevadorLlego` consiste en el sustantivo "elevador" seguido por el verbo "llegó".

En nuestra simulación crearemos una superclase llamada `EventoSimulacionElevador` (figura 11.25) que representa a un evento en nuestro modelo. Esta clase contiene una referencia `Ubicacion` (línea 11) que

representa la ubicación en la que se generó el evento, y una referencia `Object` (línea 14) que apunta al origen del evento. En nuestra simulación, los objetos utilizan instancias de `EventoSimulacionElevador` para enviar eventos a otros objetos. Cuando un objeto recibe un evento, ese objeto puede utilizar el método `obtenerUbicacion` (líneas 31 a 34) y el método `obtenerOrigen` (líneas 43 a 46) para determinar la ubicación y origen del evento.

```
1  // EventoSimulacionElevador.java
2  // Paquete de eventos básicos en la simulación del Elevador
3  package com.deitel.cpej5.elevador.evento;
4
5  // paquetes de Deitel
6  import com.deitel.cpej5.elevador.modelo.*;
7
8  public class EventoSimulacionElevador {
9
10    // Ubicación en la que se generó EventoSimulacionElevador
11    private Ubicacion ubicacion;
12
13    // objeto de origen que generó el EventoSimulacionElevador
14    private Object origen;
15
16    // el constructor de EventoSimulacionElevador establece la ubicacion
17    public EventoSimulacionElevador(Object origen,
18    Ubicacion ubicacion)
19    {
20       establecerOrigen( origen );
21       establecerUbicacion( ubicacion );
22    }
23
24    // establecer Ubicacion del EventoSimulacionElevador
25    public void establecerUbicacion( Ubicacion ubicacionEvento )
26    {
27       ubicacion = ubicacionEvento;
28    }
29
30    // obtener Ubicacion del EventoSimulacionElevador
31    public Ubicacion obtenerUbicacion()
32    {
33       return ubicacion;
34    }
35
36    // establecer origen del EventoSimulacionElevador
37    private void establecerOrigen( Object origenEvento )
38    {
39       origen = origenEvento;
40    }
41
42    // obtener origen del EventoSimulacionElevador
43    public Object obtenerOrigen()
44    {
45       return origen;
46    }
47 }
```

Figura 11.25 La clase `EventoSimulacionElevador` es la superclase de todas las demás clases de eventos en nuestro modelo.

Por ejemplo, una `Puerta` puede enviar un `EventoSimulacionElevador` a una `Persona` al abrirse o cerrarse, y el `Elevador` puede enviar un `EventoSimulacionElevador` para informar a una persona sobre su llegada o partida. Hacer que distintos objetos envíen el mismo tipo de evento para describir distintas acciones puede ser algo confuso. Para eliminar ambigüedades mientras hablamos sobre cuáles eventos enviarán los objetos, en la figura 11.26 creamos varias subclases de `EventoSimulacionElevador` para que sea más fácil asociar cada evento con su emisor. De acuerdo con la figura 11.26, las clases `EventoTimbre`, `EventoMoverPersona`, `EventoLuz`, `EventoBoton`, `EventoMoverElevador` y `EventoPuerta` son subclases de la clase `EventoSimulacionElevador`. Al utilizar estas subclases de eventos, una `Puerta` envía un evento distinto (un `EventoPuerta`) al de un `Boton` (`EventoBoton`). En la figura 11.27 se muestran las acciones que desencadenan los eventos de las subclases. Observe que todas las acciones en la figura 11.27 aparecen en la forma "sustantivo" + "verbo".

Manejo de eventos

El manejo de eventos en Java es similar al concepto de una *colaboración*, que se describe en la sección 7.10. El manejo de eventos consiste en que un objeto de una clase envía un mensaje específico (un evento) a objetos de

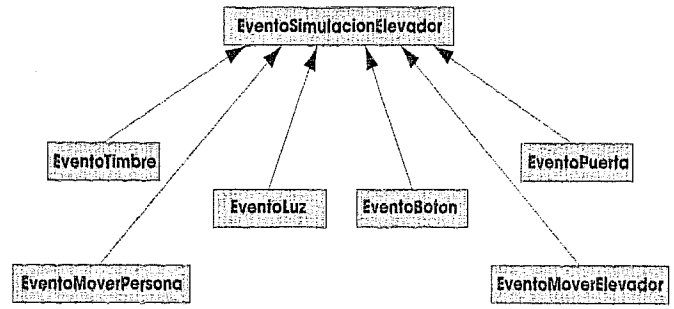


Figura 11.26 Diagrama de clases que modela la generalización entre `EventoSimulacionElevador` y sus subclases.

Evento	Se envía cuando (acción que lo desencadena)	Enviado por un objeto de la clase
`EventoTimbre`	el `Timbre` ha sonado	`Timbre`
`EventoBoton`	un `Boton` se ha oprimido	`Boton`
	un `Boton` se ha restablecido	`Boton`
`EventoPuerta`	una `Puerta` se ha abierto	`Puerta`
	una `Puerta` se ha cerrado	`Puerta`
`EventoLuz`	una `Luz` se ha encendido	`Luz`
	una `Luz` se ha apagado	
`EventoMoverPersona`	una `Persona` ha sido creada	`Persona`
	una `Persona` ha llegado al `Elevador`	
	una `Persona` ha entrado al `Elevador`	
	una `Persona` ha salido del `Elevador`	
	una `Persona` ha oprimido un `Boton`	
	una `Persona` ha salido de la simulación	
`EventoMoverElevador`	el `Elevador` ha llegado a un `Piso`	`Elevador`
	el `Elevador` ha salido de un `Piso`	

Figura 11.27 Acciones desencadenantes de los eventos de las subclases de `EventoSimulacionElevador`.

http://librosysolucionarios.net

otras clases *que están a la espera de escuchar ese tipo de mensaje*.[2] La diferencia es que los objetos que reciben el mensaje deben *registrarse* para recibirlo; por lo tanto, el manejo de eventos describe *cómo* un objeto envía un evento a otros objetos a la espera de "escuchar" ese tipo de evento; a estos objetos se les conoce como *componentes de escucha de eventos*. Para enviar un evento, el objeto emisor invoca a un método específico del objeto receptor y le pasa el objeto evento como parámetro. En nuestra simulación, este objeto evento pertenece a una clase que extiende a `EventoSimulacionElevador`.

En la figura 7.19 presentamos un diagrama de colaboración, en el que se muestran las interacciones de dos objetos `Persona` (`pasajeroEsperando` y `pasajeroViajando`) a medida que entran y salen del `Elevador`. En la figura 11.28 se muestra un diagrama modificado, en el que se incorpora el manejo de eventos. Hay dos diferencias entre los diagramas. En primer lugar, las interacciones de la figura 11.28 ocurren en el primer `Piso`. Esto nos permite nombrar a todos los objetos `Boton` y `Puerta` (`puertaPrimerPiso` y `botonPrimerPiso`) para eliminar la ambigüedad, ya que las clases `Boton` y `Puerta` tienen cada una tres objetos en nuestra simulación. Las interacciones que ocurren en el segundo `Piso` son idénticas a las que ocurren en el primer `Piso`.

La segunda diferencia (y la más substancial) entre las figuras 11.28 y 7.19 es que el `Elevador` *informa* a cada objeto (por medio de un evento) de una acción *que acaba de ocurrir*: el `Elevador` *ha llegado*. Los objetos que reciben el evento realizan entonces cierta acción en respuesta al tipo de mensaje que reciben.

De acuerdo con los mensajes `1`, `2`, `3` y `4`, el `Elevador` realiza solamente una acción: envía eventos `elevadorLlego` a los objetos interesados en recibir esos eventos. Específicamente, el objeto `Elevador` envía un `EventoMoverElevador` utilizando el método `elevadorLlego` del objeto receptor. La figura 11.28 comienza con el `Elevador` enviando un evento `elevadorLlego` al `botonElevador`. A continuación, el `botonElevador` se *restablece a sí mismo* (mensaje `1.1`). Acto seguido, el `Elevador` envía un evento `elevadorLlego` al `Timbre` (mensaje `2`), y éste a su vez invoca a su método `sonarTimbre` (es decir, el objeto `Timbre` se envía *a sí mismo* un mensaje `sonarTimbre` en el mensaje `2.1`).

Después, el `Elevador` envía un mensaje `elevadorLlego` a la `puertaElevador` (mensaje `3`). La `puertaElevador` se abre a sí misma invocando a su método `abrirPuerta` (mensaje `3.1`). En este punto la `puertaElevador` está abierta, pero no ha informado al `pasajeroViajando` de su apertura. Antes de informar al `pasajeroViajando`, la `puertaElevador` abre la `puertaPrimerPiso` enviándole un mensaje `abrirPuerta` (mensaje `3.2`); esto garantiza que el `pasajeroViajando` no salga antes de que se abra la `puertaPrimerPiso`. A continuación, la `puertaPrimerPiso` informa al `pasajeroEsperando` que se ha abierto (mensaje `3.2.1`) y el `pasajeroEsperando` entra al `Elevador` (mensaje `3.2.1.1`). Todos los mensajes anidados en `3.2` se han pasado, por lo que la `puertaElevador` puede informar al `pasajeroViajando` que la `puertaElevador` se ha abierto mediante la invocación al método `puertaAbierta` del `pasajeroViajando` (mensaje `3.3`). El `pasajeroViajando` responde saliendo del `Elevador` mensaje `3.3.1`).[3]

Por último, el `Elevador` informa al `ConductoElevador` de su llegada (mensaje `4`). El `ConductoElevador` a su vez informa al `botonPrimerPiso` de la llegada (mensaje `4.1`) y el `botonPrimerPiso` se restablece a sí mismo (mensaje `4.1.1`). A continuación, el `ConductoElevador` informa a la `luzPrimerPiso` de la llegada (mensaje `4.2`), y la `luzPrimerPiso` se ilumina a sí misma (mensaje `4.2.1`).

Componentes de escucha de eventos

Ya demostramos el manejo de eventos entre el `Elevador` y el objeto `puertaElevador` utilizando el diagrama de colaboración modificado de la figura 11.28; el `Elevador` envía un evento `elevadorLlego` a la `puertaElevador` (mensaje `3`). Primero debemos determinar el objeto evento que el `Elevador` pasará a la `puertaElevador`. De acuerdo con la nota en la esquina inferior izquierda de la figura 11.28, el `Elevador` pasa un objeto `EventoMoverElevador` (figura 11.29) cuando invoca a un método `elevadorLlego`. El

2. Técnicamente, un objeto envía una notificación de un evento (o cierta acción desencadenadora) a otro objeto. Sin embargo, el dialecto de Java se refiere a enviar esta notificación como "enviar un evento".
3. El diagrama de colaboración sigue teniendo el problema de que el `pasajeroEsperando` entre al `Elevador` (mensaje `3.2.1.1`) antes de que salga el `pasajeroViajando` (mensaje `3.3.1`). En la sección 16.12 veremos cómo resolver este problema mediante el uso del subprocesamiento múltiple, la sincronización y las clases activas.

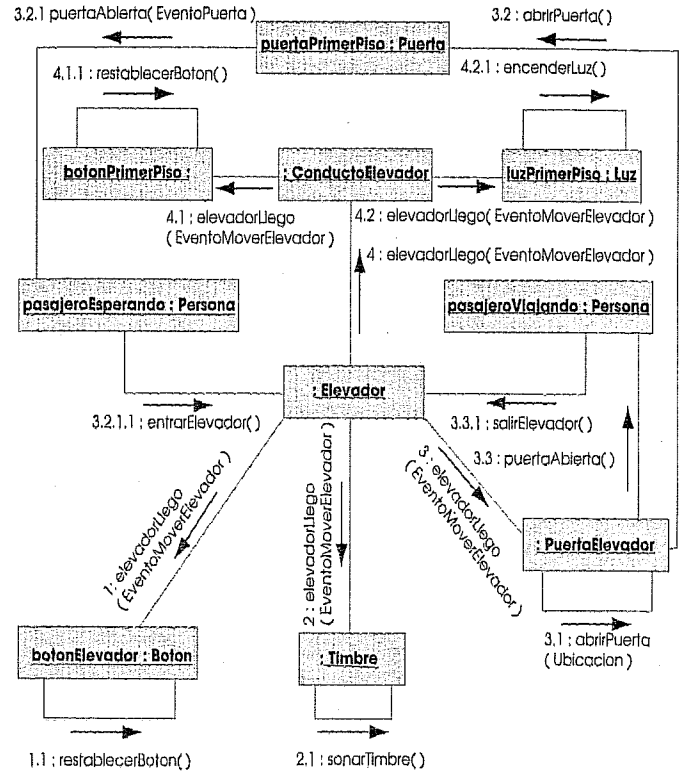


Figura 11.28 Diagrama de colaboración modificado para los pasajeros que entran y salen del `Elevador` en el primer `Piso`.

diagrama de clases de generalización de la figura 11.26 indica que `EventoMoverElevador` es una subclase de `EventoSimulacionElevador`, por lo que `EventoMoverElevador` hereda las referencias `Object` y `Ubicacion` de `EventoSimulacionElevador`.[4]

La `puertaElevador` debe implementar una interfaz que "escuche" por si hay un `EventoMoverElevador`; esto convierte a la `puertaElevador` en un componente de escucha de eventos. La interfaz `EscuchaMoverElevador` (figura 11.30) proporciona los métodos `elevadorPartio` (línea 8) y `elevadorLlego` (línea 11), los cuales permiten al `Elevador` notificar al `EscuchaMoverElevador` cuando el `Elevador` ha llegado o partido. A una interfaz que proporciona los métodos para un componente de escucha de eventos como `EscuchaMoverElevador`, se le conoce como *interfaz de escucha de eventos*.

Los métodos `elevadorLlego` y `elevadorPartio` reciben cada uno un objeto `EventoMoverElevador` (figura 11.29) como argumento. Por lo tanto, cuando el `Elevador` "envía un evento `elevadorLlego`" a otro objeto, el `Elevador` pasa un objeto `EventoMoverElevador` como argumento para el

4. En nuestra simulación, todas las clases de eventos tienen esta estructura; es decir, la estructura de la clase `EventoMoverElevador` es idéntica a la estructura de las clases `EventoPuerta`, `EventoBoton`, etcétera. Cuando termine de leer esta sección, le recomendamos que vea la implementación de los eventos en el apéndice D para obtener más detalles sobre la estructura de eventos de nuestro sistema; en las figuras D.1 a D.7 se presenta el código para los eventos, y en las figuras D.8 a D.14 se presenta el código para los componentes de escucha de los eventos.

http://librosysolucionarios.net

```
1  // EventoMoverElevador.java
2  // Indica a qué Piso llegó, o de cual Piso salió el Elevador
3  package com.deitel.cpej5.elevador.evento;
4
5  // paquetes de Deitel
6  import com.deitel.cpej5.elevador.modelo.*;
7
8  public class EventoMoverElevador extends EventoSimulacionElevador {
9
10    // constructor de EventoMoverElevador
11    public EventoMoverElevador( Object origen, Ubicacion ubicacion )
12    {
13       super( origen, ubicacion );
14    }
15 }
```

Figura 11.29 La clase **EventoMoverElevador**, una subclase de **EventoSimulacionElevador**, se envía cuando el **Elevador** ha llegado o partido de un **Piso**.

método **elevadorLlego** del objeto receptor. Implementaremos la clase **Puerta** (la clase de la cual **puertaElevador** es una instancia) en el apéndice H, después de seguir refinando nuestro diseño y aprendiendo más herramientas de Java.

Revisión del diagrama de clases

En la figura 11.31 se modifican las asociaciones del diagrama de clases de la figura 10.28, para incluir el manejo de eventos. Al igual que en el diagrama de colaboración de la figura 11.28, en la figura 11.31 se indica que un objeto informa (o *indica*) a otro objeto que ha ocurrido un evento. Si un objeto que recibe el evento invoca a un método **private**, el diagrama de clases representa esta invocación al método como una *autoasociación* (es decir, la clase contiene una asociación consigo misma). Las clases **Boton**, **Puerta**, **Luz** y **Timbre** contienen autoasociaciones; observe que la asociación no incluye una punta de flecha que indica la dirección de la asociación, ya que la asociación de la clase es consigo misma. Por último, el diagrama incluye una asociación entre la clase **Puerta** y la clase **Persona** (la **Puerta** informa a una **Persona** que se ha abierto), ya que establecimos la relación entre todos los objetos **Puerta** y un objeto **Persona**.

```
1  // EscuchaMoverElevador.java
2  // Métodos invocados cuando Elevador ha partido o llegado
3  package com.deitel.cpej5.elevador.evento;
4
5  public interface EscuchaMoverElevador {
6
7     // se invoca cuando el Elevador ha partido
8     public void elevadorPartio( EventoMoverElevador eventoMover );
9
10    // se invoca cuando el Elevador ha llegado
11    public void elevatorArrived( EventoMoverElevador eventoMover );
12 }
```

Figura 11.30 La interfaz **EscuchaMoverElevador** proporciona los métodos requeridos para escuchar eventos de partida y llegada del **Elevador**.

http://librosysolucionarios.net

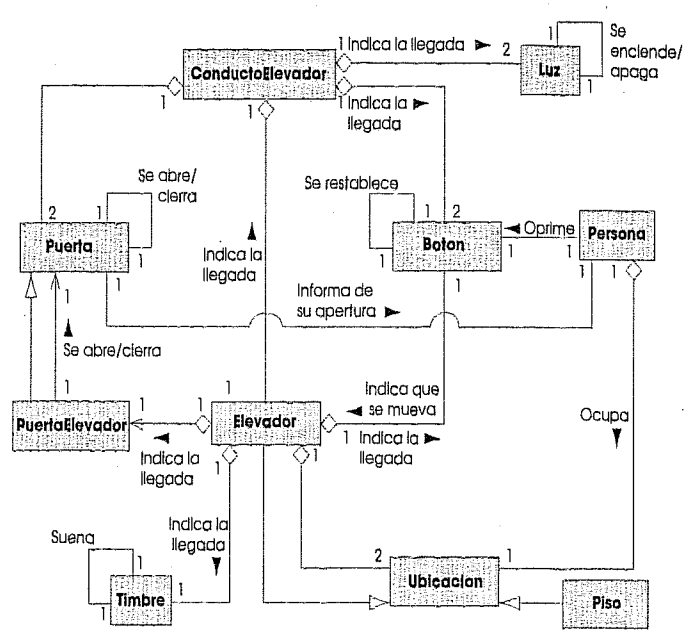


Figura 11.31 Diagrama de clases de nuestro simulador (incluyendo el manejo de eventos).

RESUMEN

- El valor de una literal de carácter es su valor entero en el conjunto de caracteres Unicode. Las cadenas pueden incluir letras, dígitos y caracteres especiales como +, -, *, / y $. Una cadena en Java es un objeto de la clase **String**. Las literales de cadena (constantes de cadena) se conocen comúnmente como objetos **String** anónimos, y se escriben entre comillas dobles en los programas.
- El método **length** de la clase **String** devuelve el número de caracteres en un objeto **String**.
- El método **charAt** de la clase **String** devuelve el carácter que está en una posición específica.
- El método **equals** compara la igualdad entre dos objetos. Este método devuelve **true** si los objetos son iguales y **false** en caso contrario. El método **equals** utiliza una comparación lexicográfica para los objetos **String**.
- Cuando los valores de tipo primitivo se comparan con ==, el resultado es **true** si ambos valores son idénticos. Cuando las referencias se comparan con ==, el resultado es **true** si ambas referencias se refieren al mismo objeto en memoria.
- Java trata a todas las literales de cadena con el mismo contenido que un objeto **String** anónimo.
- El método **equalsIgnoreCase** de la clase **String** realiza una comparación que no toma en cuenta el uso de las mayúsculas/minúsculas.
- El método **compareTo** de la clase **String** utiliza una comparación lexicográfica y devuelve 0 si las cadenas que compara son iguales, un número negativo si la cadena desde la que se invoca **CompareTo** es menor que la cadena que se pasa como argumento, y un número positivo si la cadena desde la que se invoca **CompareTo** es mayor que la cadena que se pasa como argumento.
- El método **regionMatches** de la clase **String** compara porciones de dos cadenas para ver si son iguales.
- El método **startsWith** de la clase **String** determina si una cadena empieza con los caracteres que se especifican como argumento. El método **endsWith** de la clase **String** determina si una cadena termina con los caracteres que se especifican como argumento.
- El método **indexOf** de la clase **String** localiza la primera ocurrencia de un carácter o subcadena en una cadena. El método **lastIndexOf** localiza la última ocurrencia de un carácter o subcadena en una cadena.
- El método **substring** de la clase **String** copia y devuelve parte de un objeto cadena existente.
- El método **concat** de la clase **String** concatena dos objetos cadena y devuelve un nuevo objeto cadena que contiene los caracteres de ambas cadenas originales.

- El método `replace` de la clase `String` devuelve un nuevo objeto cadena que reemplaza todas las ocurrencias en un objeto `String` de su primer argumento carácter, con su segundo argumento carácter.
- El método `toUpperCase` de la clase `String` devuelve una nueva cadena con letras mayúsculas en las posiciones en las que la cadena original tenía letras minúsculas. El método `toLowerCase` devuelve una nueva cadena con letras minúsculas en las posiciones en las que la cadena original tenía letras mayúsculas.
- El método `trim` de la clase `String` devuelve un nuevo objeto cadena, en el que todos los caracteres de espacio en blanco (como espacios, nuevas líneas y tabuladores) se han eliminado del principio y del final de una cadena.
- El método `toCharArray` de la clase `String` devuelve un arreglo `char` que contiene una copia de los caracteres de la cadena.
- El método `valueOf` de la clase `String` devuelve su argumento convertido en una cadena.
- La clase `StringBuffer` proporciona tres constructores que permiten la inicialización de objetos `StringBuffer` sin caracteres y con una capacidad inicial de 16 caracteres; sin caracteres y con una capacidad inicial especificada en el argumento entero; o con una copia de los caracteres del argumento `String` y una capacidad inicial equivalente al número de caracteres en el argumento `String` más 16.
- El método `length` de la clase `StringBuffer` devuelve el número de caracteres que están almacenados en un objeto `StringBuffer`. El método `capacity` devuelve el número de caracteres que pueden almacenarse en un objeto `StringBuffer` sin tener que asignar más memoria.
- El método `ensureCapacity` asegura que un objeto `StringBuffer` tenga cuando menos la capacidad especificada. El método `setLength` incrementa o decrementa la longitud de un objeto `StringBuffer`.
- El método `charAt` de la clase `StringBuffer` devuelve el carácter que se encuentra en el índice especificado. El método `setCharAt` establece el carácter en la posición especificada. El método `getChars` copia caracteres del objeto `StringBuffer` al arreglo de caracteres que se pasa como argumento.
- La clase `StringBuffer` proporciona métodos `append` sobrecargados para agregar valores de tipo primitivo, arreglo de caracteres, `String` y `Object` al final de un objeto `StringBuffer`. Los objetos de esta clase y los métodos `append` son utilizados por el compilador de Java para implementar los operadores de concatenación + y +=.
- La clase `StringBuffer` proporciona los métodos `insert` sobrecargados para insertar valores de tipo primitivo, arreglo de caracteres, `String` y `Object` en cualquier posición de un objeto `StringBuffer`.
- La clase `Character` proporciona un constructor que toma un argumento carácter.
- El método `isDefined` de la clase `Character` determina si un carácter está definido en el conjunto de caracteres Unicode. De ser así, el método devuelve `true`, en caso contrario devuelve `false`.
- El método `isDigit` de la clase `Character` determina si un carácter es un dígito definido en Unicode. De ser así, el método devuelve `true`, en caso contrario devuelve `false`.
- El método `isJavaIdentifierStart` de la clase `Character` determina si un carácter puede utilizarse como el primer carácter de un identificador en Java [es decir, una letra, un guión bajo (_) o un signo de dólares ($)]. De ser así, el método devuelve `true`, en caso contrario devuelve `false`.
- El método `isJavaIdentifierPart` de la clase `Character` determina si un carácter puede utilizarse en un identificador en Java [es decir, una letra, un guión bajo (_) o un signo de dólares ($)]. El método `isLetterOrDigit` determina si un carácter es una letra o un dígito. En cada caso, si se cumple la condición el método devuelve `true`, en caso contrario devuelve `false`.
- El método `isLowerCase` de la clase `Character` determina si un carácter es letra minúscula. El método `isUpperCase` determina si un carácter es una letra mayúscula. En ambos casos, si se cumple la condición el método devuelve `true`, en caso contrario devuelve `false`.
- El método `toUpperCase` de la clase `Character` convierte un carácter en su equivalente en mayúscula. El método `toLowerCase` convierte un carácter en su equivalente en minúscula.
- El método `digit` de la clase `Character` convierte su argumento carácter en un entero, en el sistema numérico especificado por su argumento entero `raiz`. El método `forDigit` convierte su argumento entero `digito` en un carácter del sistema numérico especificado por su argumento entero `raiz`.
- El método `charValue` de la clase `Character` devuelve el valor `char` almacenado en un objeto `Character`. El método `toString` devuelve una representación `String` de un objeto `Character`.
- El constructor predeterminado de `StringTokenizer` crea un objeto `StringTokenizer` para su argumento de cadena que utiliza la cadena delimitadora predeterminada `"\t\n\r\f"`, que consiste en un espacio, una nueva línea, un tabulador y un retorno de carro para la descomposición de una cadena en tokens.
- El método `countTokens` de la clase `StringTokenizer` devuelve el número de tokens en una cadena que va a descomponerse en tokens.
- El método `hasMoreTokens` de la clase `StringTokenizer` determina si hay más tokens en la cadena que se está descomponiendo en tokens.
- El método `nextToken` de la clase `StringTokenizer` devuelve un objeto `String` con el siguiente token.

TERMINOLOGÍA

anexar cadenas a otras cadenas
`append`, método de la clase `StringBuffer`
arreglo de caracteres
cadena
cadena de búsqueda
cadena vacía
`capacity`, método de la clase `StringBuffer`
caracteres de espacio en blanco
`Character`, clase
`charAt`, método de la clase `String`
`charAt`, método de la clase `StringBuffer`
`charValue`, método de la clase `Character`
código de caracteres
comparación de cadenas
`compareTo`, método de la clase `String`
`concat`, método de la clase `String`
concatenación
concatenación de cadenas
conjunto de caracteres
`countTokens`, método (`StringTokenizer`)
delimitador
descomponer cadenas en tokens
`digit`, método de la clase `Character`
dígitos hexadecimales
`endsWith`, método de la clase `String`
`equals`, método de la clase `String`
`equalsIgnoreCase`, método de `String`
`forDigit`, método de la clase `Character`
`getChars`, método de la clase `String`
`getChars`, método de la clase `StringBuffer`
`hasMoreTokens`, método
`indexOf`, método de la clase `String`
`insert`, método de la clase `StringBuffer`
`isDefined`, método de la clase `Character`
`isDigit`, método de la clase `Character`
`isJavaIdentifierPart`, método
`isJavaIdentifierStart`, método
`isLetter`, método de la clase `Character`
`isLetterOrDigit`, método de `Character`
`isLowerCase`, método de la clase `Character`
`isUpperCase`, método de la clase `Character`
`lastIndexOf`, método de la clase `String`
`length`, método de la clase `String`
`length`, método de la clase `StringBuffer`
literal
literal de cadena
literal de carácter
longitud de una cadena
`nextToken`, método de `StringTokenizer`
procesamiento de cadenas
`regionMatches`, método de la clase `String`
`replace`, método de la clase `String`
representación de un carácter en código numérico
`setCharAt`, método de la clase `StringBuffer`
`startsWith`, método de la clase `String`
`String`, clase
`StringBuffer`, clase
`StringIndexOutOfBounds-Exception`
`StringTokenizer`, clase
`substring`, método de la clase `String`
`toCharArray`, método de la clase `String`
token
`toLowerCase`, método de la clase `Character`
`toLowerCase`, método de la clase `String`
`toString`, método de la clase `Character`
`toString`, método de la clase `StringBuffer`
`toUpperCase`, método de la clase `Character`
`toUpperCase`, método de la clase `String`
`trim`, método de la clase `String`
Unicode, conjunto de caracteres
`valueOf`, método de la clase `String`

EJERCICIOS DE AUTOEVALUACIÓN

11.1 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Cuando los objetos `String` se comparan utilizando `==`, el resultado es `true` si los objetos `String` contiene los mismos valores.
b) Un objeto `String` puede modificarse una vez creado.

11.2 Para cada uno de los siguientes enunciados, escriba una instrucción que realice la tarea indicada:
a) Comparar la cadena en `s1` con la cadena en `s2` para ver si su contenido es igual.
b) Anexar la cadena `s2` a la cadena `s1`, utilizando `+=`.
c) Determinar la longitud de la cadena en `s1`.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

11.1 a) Falso. Los objetos `String` se comparan con el operador `==` para determinar si son el mismo objeto en la memoria.
b) Falso. Los objetos `String` son inmutables y no pueden modificarse una vez creados. Los objetos `StringBuffer` sí pueden modificarse una vez creados.

http://librosysolucionarios.net

11.2 a) `s1.equals( s1 )`
b) `s1 += s1;`
c) `s1.length()`

EJERCICIOS

Los ejercicios 11.3 a 11.6 son razonablemente complicados. Una vez que haya resuelto estos problemas, podrá implementar fácilmente la mayoría de los juegos de cartas.

11.3 Modifique el programa de la figura 11.9, de manera que el método para repartir cartas reparta una mano de póker de cinco cartas. Después escriba métodos para determinar si la mano contiene:
a) un par;
b) dos pares;
c) tres cartas iguales (por ejemplo, tres jotos);
d) cuatro cartas iguales (por ejemplo, cuatro ases);
e) un flux (es decir, cinco cartas del mismo palo);
f) una escalera (es decir, cinco cartas de valores consecutivos de una cara);
g) un full (es decir, dos cartas de un valor de una cara y tres cartas de un valor de otra cara).

11.4 Utilice los métodos desarrollados en el ejercicio 11.3 para escribir un programa que reparta dos manos de póker de cinco cartas, que evalúe cada mano y determine cuál es la mejor.

11.5 Modifique el programa desarrollado en el ejercicio 11.4, de manera que se pueda simular el repartidor. La mano de cinco cartas del repartidor se reparte "boca abajo", de manera que el jugador no pueda verla. El programa deberá entonces evaluar la mano del repartidor y, con base en la calidad de ésta, el repartidor deberá sacar una, dos o tres cartas más para sustituir el número correspondiente de cartas innecesarias en la mano original. El programa deberá entonces reevaluar la mano del repartidor. (*Precaución:* ¡Éste es un problema difícil!)

11.6 Modifique el programa desarrollado en el ejercicio 11.5 para que pueda manejar la mano del repartidor automáticamente, pero el jugador podrá decidir qué cartas va a sustituir de su mano. El programa deberá entonces evaluar ambas manos para determinar quién será el ganador. Ahora, utilice este nuevo programa para jugar 20 veces contra la computadora. ¿Quién ganó más juegos, usted o la computadora? Haga que un amigo juegue 20 veces contra la computadora. ¿Quién ganó más juegos? Con base en los resultados de estos juegos, refine su programa para jugar póker. (Éste también es un problema difícil.) Juegue 20 veces más. ¿Su programa modificado juega mejor?

11.7 Escriba una aplicación que utilice el método `compareTo` de la clase `String` para comparar dos cadenas introducidas por el usuario. Muestre si la primera cadena es menor, igual o mayor que la segunda.

11.8 Escriba una aplicación que utilice el método `regionMatches` de la clase `String` para comparar dos cadenas introducidas por el usuario. El programa deberá recibir también como entrada el número de caracteres a comparar y el índice inicial de la comparación. El programa deberá indicar si las cadenas son iguales. Ignore si los caracteres están en mayúsculas o en minúsculas al momento de realizar la comparación.

11.9 Escriba una aplicación que utilice la generación de números aleatorios para crear enunciados. Use cuatro arreglos de cadenas llamados `articulo`, `sustantivo`, `verbo` y `preposicion`. Cree una oración seleccionando una palabra al azar de cada uno de los arreglos, en el siguiente orden: `articulo`, `sustantivo`, `verbo`, `preposicion`, `articulo` y `sustantivo`. A medida que se elija cada palabra, concaténela con las palabras anteriores en el enunciado. Las palabras deberán separarse mediante espacios. Cuando se muestre el enunciado final, deberá empezar con una letra mayúscula y terminar con un punto. El programa deberá generar 20 enunciados y mostrarlos en un área de texto.

El arreglo de artículos debe contener los artículos `"el"`, `"un"`, `"algún"` y `"ningún"`; el arreglo de sustantivos deberá contener los sustantivos `"niño"`, `"señor"`, `"perro"`, `"ciudad"` y `"auto"`; el arreglo de verbos deberá contener los verbos `"manejó"`, `"saltó"`, `"corrió"`, `"caminó"` y `"pasó"`; el arreglo de preposiciones deberá contener las preposiciones `"a"`, `"desde"`, `"encima de"`, `"debajo de"` y `"sobre"`.

Una vez que escriba el programa anterior, modifíquelo para producir una historia breve que consista de varias de estas oraciones.

11.10 *(Quintillas.)* Una quintilla es un verso humoroso de cinco líneas, en el cual la primera y segunda línea riman con la quinta, y la tercera línea rima con la cuarta. Utilizando técnicas similares a las desarrolladas en el ejercicio 11.9, escriba un programa en Java que produzca quintillas al azar. Pulir el programa para producir buenas quintillas es un problema desafiante, ¡pero el resultado valdrá la pena!

11.11 *(Latín cerdo.)* Escriba una aplicación que codifique frases en español a frases en latín cerdo. El latín cerdo es una forma de lenguaje codificado. Existen muchas variaciones en los métodos utilizados para formar frases en latín cerdo. Por cuestiones de simpleza, utilice el siguiente algoritmo:

Para formar una frase en latín cerdo a partir de una frase en español, divida la frase en palabras con un objeto de la clase `StringTokenizer`. Para traducir cada palabra en español a una palabra en latín cerdo, coloque la primera letra de la palabra en español al final de la palabra, y agregue las letras `"ae"`. De esta forma, la palabra `"salta"` se convierte a `"altasae"`, la palabra `"el"` se convierte en `"leae"` y la palabra `"computadora"` se convierte en `"omputadoracae"`. Los espacios en blanco entre las palabras permanecen como espacios en blanco. Suponga que la frase en español consiste en palabras separadas por espacios en blanco, que no hay signos de puntuación y que todas las palabras tienen dos o más letras. El método `imprimirPalabraEnLatin` deberá mostrar cada palabra. Cada token devuelto de `nextToken` se pasará al método `imprimirPalabraEnLatin` para imprimir la palabra en latín cerdo. Permita al usuario introducir el enunciado. Use un área de texto para ir mostrando cada uno de los enunciados convertidos.

11.12 Escriba una aplicación que reciba como entrada un número telefónico como una cadena de la forma `(555) 555-5555`. El programa deberá utilizar un objeto de la clase `StringTokenizer` para extraer el código de área como un token, los primeros tres dígitos del número telefónico como otro token y los últimos cuatro dígitos del número telefónico como otro token. Los siete dígitos del número telefónico deberán concatenarse en una cadena. El programa deberá convertir la cadena del código de área en `int` (¡recuerde utilizar `parseInt`!) y deberá convertir la cadena del número telefónico en `long`. Deberán imprimirse tanto el código de área como el número telefónico. Recuerde que tendrá que modificar los caracteres delimitadores durante el proceso de dividir la cadena en tokens.

11.13 Escriba una aplicación que reciba como entrada una línea de texto, que divida la línea en tokens mediante un objeto de la clase `StringTokenizer` y que muestre los tokens en orden inverso. Use caracteres de espacio como delimitadores.

11.14 Use los métodos de comparación de cadenas que se describieron en este capítulo, junto con las técnicas para ordenar arreglos que se desarrollaron en el capítulo 7, para escribir un programa que ordene alfabéticamente una lista de cadenas. Permita al usuario introducir las cadenas en un campo de texto. Muestre los resultados en un área de texto.

11.15 Escriba una aplicación que reciba como entrada una línea de texto y que lo imprima dos veces; una vez en letras mayúsculas y otra en letras minúsculas.

11.16 Escriba una aplicación que reciba como entrada varias líneas de texto y un carácter de búsqueda, y que utilice el método `indexOf` de la clase `String` para determinar el número de ocurrencias de ese carácter en el texto.

11.17 Escriba una aplicación con base en el programa del ejercicio 11.16, que reciba como entrada varias líneas de texto y utilice el método `indexOf` de la clase `String` para determinar el número total de ocurrencias de cada letra del alfabeto en ese texto. Las letras mayúscula y minúscula deben contarse como una sola. Almacene los totales para cada letra en un arreglo e imprima los valores en formato tabular, después de que se hayan determinado los totales.

11.18 Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con la letra "b". Los resultados deberán aparecer en un área de texto.

11.19 Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con las letras "ed". Los resultados deberán aparecer en un área de texto.

11.20 Escriba una aplicación que reciba como entrada un código entero para un carácter, y que muestre el carácter correspondiente. Modifique este programa de manera que genere todos los posibles códigos de tres dígitos en el rango de 000 a 255, y que intente imprimir los caracteres correspondientes. Muestre los resultados en un área de texto.

11.21 Escriba sus propias versiones de los métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String`.

11.22 Escriba un programa que lea una palabra de cinco letras proveniente del usuario, y que produzca todas las posibles palabras de tres letras que puedan derivarse de las letras de la palabra con cinco letras. Por ejemplo, las palabras de tres letras producidas a partir de la palabra "trigo" son "rio", "tio" y "oir".

SECCIÓN ESPECIAL: EJERCICIOS AVANZADOS DE MANIPULACIÓN DE CADENAS

Los siguientes ejercicios son claves para el libro y están diseñados para evaluar la comprensión del lector sobre los conceptos fundamentales de la manipulación de cadenas. Esta sección incluye una colección de ejercicios intermedios y avanzados de manipulación de cadena. El lector encontrará estos ejercicios desafiantes, pero divertidos. Los problemas varían considerablemente en dificultad. Algunos requieren una hora o dos para escribir e implementar el programa. Otros son útiles como tareas de laboratorio que pudieran requerir dos o tres semanas de estudio e implementación. Algunos son proyectos de fin de curso desafiantes.

11.23 *(Análisis de textos.)* La disponibilidad de computadoras con capacidades de manipulación de cadenas ha dado como resultado algunos métodos interesantes para analizar los escritos de grandes autores. Se ha puesto mucha atención en saber si realmente vivió William Shakespeare. Algunos estudiosos creen que existe evidencia importante que indica que, en realidad fueron Christopher Marlowe u otros autores quienes escribieron las obras maestras que

http://librosysolucionarios.net

se atribuyen a Shakespeare. Los investigadores han utilizado computadoras para buscar similitudes en los escritos de estos dos autores. En este ejercicio se examinan tres métodos para analizar textos mediante una computadora.

a) Escriba una aplicación que lea varias líneas de texto desde el teclado e imprima una tabla que indique el número de ocurrencias de cada letra del alfabeto en el texto. Por ejemplo, la frase:

```
Ser o no ser: ése es el dilema:
```

contiene una "a", ninguna "b", ninguna "c", etcétera.

b) Escriba una aplicación que lea varias líneas de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras, etcétera, que aparezcan en el texto. Por ejemplo, en la figura 11.32 se muestra la cuenta para la frase:

```
¿Qué es más noble para el espíritu?
```

c) Escriba una aplicación que lea varias líneas de texto e imprima una tabla que indique el número de ocurrencias de cada palabra distinta en el texto. La primera versión de su programa debe incluir las palabras en la tabla, en el mismo orden en el cual aparecen en el texto. Por ejemplo, las líneas:

```
Ser o no ser: ése es el dilema:
¿Qué es más noble para el espíritu?
```

contiene la palabra "ser" dos veces, La palabra "o" una vez, la palabra "ése" una vez, etcétera. Una muestra más interesante (y útil) podría ser intentar con las palabras ordenadas alfabéticamente.

11.24 *(Impresión de fechas en varios formatos.)* Las fechas se imprimen en varios formatos comunes. Dos de los formatos más utilizados son:

```
25/04/1955 y 25 Abril, 1955
```

Escriba una aplicación que lea una fecha en el primer formato e imprima dicha fecha en el segundo formato.

11.25 *(Protección de cheques.)* Las computadoras se utilizan frecuentemente en sistemas de escritura de cheques tales como aplicaciones para nóminas y para cuentas por pagar. Existen muchas historias extrañas acerca de cheques de pago que se imprimen (por error) con montos que se exceden por millones. Los sistemas de escritura de cheques computarizados imprimen cantidades incorrectas debido al error humano o a una falla de la máquina. Los diseñadores de sistemas construyen controles en sus sistemas para evitar la emisión de dichos cheques erróneos.

Otro problema grave es la alteración intencional del monto de un cheque por alguien que planee cobrar un cheque de manera fraudulenta. Para evitar la alteración de un monto, la mayoría de los sistemas computarizados emplean una técnica llamada *protección de cheques*. Los cheques diseñados para impresión por computadora contienen un número fijo de espacios en los cuales la computadora puede imprimir un monto. Suponga que un cheque contiene ocho espacios en blanco en los cuales la computadora puede imprimir el monto de un cheque de nómina semanal. Si el monto es grande, entonces se llenarán los ocho espacios. Por ejemplo:

```
1,230.60    (monto del cheque)
12345678    (números de posición)
```

Por otra parte, si el monto es menor de $1,000, entonces varios espacios quedarían vacíos. Por ejemplo:

```
  99.87
12345678
```

Longitud de palabra	Ocurrencias
1	0
2	2
3	2
4	1
5	1
6	0
7	0
8	1

Figura 11.32 La cuenta de longitudes de palabras para la cadena `"¿Qué es más noble para el espíritu?"`.

contiene tres espacios en blanco. Si se imprime un cheque con espacios en blanco, es más fácil para alguien alterar el monto del cheque. Para evitar que se altere el cheque, muchos sistemas de escritura de cheques insertan *asteriscos al principio* para proteger la cantidad, como se muestra a continuación:

```
***99.87
12345678
```

Escriba una aplicación que reciba como entrada un monto a imprimir sobre un cheque y que lo escriba mediante el formato de protección de cheques, con asteriscos al principio si es necesario. Suponga que existen nueve espacios disponibles para imprimir el monto.

11.26 *(Escritura en letras del código de un cheque.)* Para continuar con la discusión del ejercicio 11.25, reiteramos la importancia de diseñar sistemas de escritura de cheques para evitar la alteración de los montos de los cheques. Un método común de seguridad requiere que el monto del cheque se escriba tanto en números como en letras. Incluso si alguien puede alterar el monto numérico del cheque, es extremadamente difícil modificar el monto en letras.

a) Muchos sistemas de escritura de cheques computarizados no imprimen el monto de los cheques en letras. Tal vez la principal razón de esta omisión sea que la mayoría de los lenguajes de alto nivel utilizados en aplicaciones comerciales no contienen las herramientas adecuadas de manipulación de caracteres. Otra razón es que la lógica para escribir las letras equivalentes al monto de los cheques es algo complicada.

b) Escriba una aplicación que reciba como entrada un monto numérico de un cheque y que escriba el monto en letras. Por ejemplo, el monto `112.43` debe escribirse así:

```
CIENTO DOCE CON 43/100
```

11.27 *(Clave Morse.)* Quizá el más famoso de todos los esquemas de codificación es el código Morse, desarrollado por Samuel Morse en 1832 para usarlo con el sistema telegráfico. El código Morse asigna una serie de puntos y guiones a cada letra del alfabeto, cada dígito y algunos caracteres especiales (tales como el punto, la coma, los dos puntos y el punto y coma). En los sistemas orientados a sonidos, el punto representa un sonido corto y el guión representa un sonido largo. Otras representaciones de puntos y guiones se utilizan en sistemas orientados a luces y sistemas de señalización con banderas. La separación entre palabras se indica mediante un espacio o, simplemente, con la ausencia de un punto o un guión. En un sistema orientado a sonidos, un espacio se indica por un tiempo breve durante el cual no se transmite sonido alguno. La versión internacional del código Morse aparece en la figura 11.33.

Escriba una aplicación que lea una frase en español y que codifique la frase en clave Morse. Además, escriba un programa que lea una frase en código Morse y que la convierta en su equivalente en español. Use un espacio en blanco entre cada letra en clave Morse y tres espacios en blanco entre cada palabra en clave Morse.

11.28 *(Programa de conversión al sistema métrico.)* Escriba una aplicación que ayude al usuario a realizar conversiones métricas. Su programa debe permitir al usuario especificar los nombres de la unidades como cadenas (es decir, centímetros, litros, gramos, etcétera, para el sistema métrico, y pulgadas, cuartos, libras, etcétera, para el sistema inglés) y debe responder a preguntas simples tales como:

```
"¿Cuántas pulgadas hay en 2 metros?"
"¿Cuántos litros hay en 10 cuartos?"
```

Su programa debe reconocer conversiones inválidas. Por ejemplo, la pregunta:

```
"¿Cuántos pies hay en 5 kilogramos?"
```

no es correcta, debido a que los `"pies"` son unidades de longitud, mientras que los `"kilogramos"` son unidades de masa.

SECCIÓN ESPECIAL: PROYECTOS DESAFIANTES DE MANIPULACIÓN DE CADENAS

11.29 *(Proyecto: un corrector ortográfico.)* Muchos paquetes populares de software de procesamiento de palabras cuentan con correctores ortográficos integrados. En este proyecto usted debe desarrollar su propia herramienta de corrección ortográfica. Le haremos unas sugerencias para ayudarlo a empezar. Sería conveniente que después le agregara más características. Use un diccionario computarizado (si tiene acceso a uno) como fuente de palabras.

¿Por qué escribimos tantas palabras en forma incorrecta? En algunos casos es porque simplemente no conocemos la manera correcta de escribirlas, por lo que tratamos de adivinar lo mejor que podemos. En otros casos, es porque transponemos dos letras (por ejemplo, "perdeterminado" en lugar de "predeterminado"). Algunas veces escribimos una letra doble por accidente (por ejemplo, "úttil" en vez de "útil"). Otras veces escribimos una tecla que está cerca de la que pretendíamos escribir (por ejemplo, "cunpleaños" en vez de "cumpleaños"), y así sucesivamente.

http://librosysolucionarios.net

Carácter	Código	Carácter	Código
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	*Dígitos*	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Figura 11.33 Las letras del alfabeto expresadas en código Morse Internacional.

Diseñe e implemente una aplicación de corrección ortográfica en Java. Su programa debe mantener un arreglo de cadenas llamado `listaDePalabras`. Permita al usuario introducir estas cadenas. [*Nota:* En el capítulo 17 presentaremos el procesamiento de archivos. Una vez que tenga esta capacidad, podrá obtener las palabras para el corrector ortográfico de un diccionario computarizado almacenado en un archivo.]

Su programa debe pedir al usuario que introduzca una palabra. El programa debe entonces buscar esa palabra en el arreglo `listaDePalabras`. Si la palabra se encuentra en el arreglo, su programa deberá imprimir "`La palabra está escrita correctamente`". Si la palabra no se encuentra en el arreglo, su programa debe imprimir "`La palabra no está escrita correctamente`". Después su programa debe tratar de localizar otras palabras en la `listaDePalabras` que puedan ser la palabra que el usuario trataba de escribir. Por ejemplo, puede probar con todas las transposiciones simples posibles de letras adyacentes para descubrir que la palabra "predeterminado" concuerda directamente con una palabra en `listaDePalabras`. Desde luego que esto implica que su programa comprobará todas las otras transposiciones posibles, tales como "rpedeterminado", "perdeterminado", "predetreminado", "predetemrinado" y "predetermniado". Cuando encuentre una nueva palabra que concuerde con una en la `listaDePalabras`, imprima esa palabra en un mensaje tal como "`¿Quiso usted decir "predeterminado"?`".

Implemente otras pruebas, tales como reemplazar cada letra doble con una sola letra y cualquier otra prueba que pueda desarrollar para aumentar el valor de su corrector ortográfico.

11.30 *(Proyecto: un generador de crucigramas.)* La mayoría de las personas han resuelto crucigramas, pero pocos han intentado generar uno. Aquí lo sugerimos como un proyecto de manipulación de cadenas que requiere una cantidad considerable de sofisticación y esfuerzo.

Hay muchas cuestiones que el programador tiene que resolver para hacer que funcione incluso hasta el programa generador de crucigramas más simple. Por ejemplo, ¿cómo representaría la cuadrícula de un crucigrama dentro de la computadora? ¿Debería utilizar una serie de cadenas o arreglos bidimensionales?

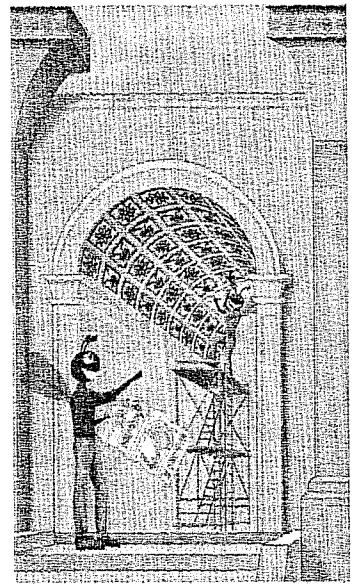
El programador necesita una fuente de palabras (es decir, un diccionario computarizado) a las que el programa pueda hacer referencia de manera directa. ¿De qué manera deben almacenarse estas palabras para facilitar las complejas manipulaciones requeridas por el programa?

Si usted es realmente ambicioso, querrá generar la porción de "claves" del crucigrama, en la que se imprimen las breves pistas para cada palabra "horizontal" y cada palabra "vertical". La sola impresión del crucigrama [illegible] co no es un problema sencillo.

http://librosysolucionarios.net

12

Gráficos y Java2D



Objetivos

- Comprender los contextos de gráficos y los objetos de gráficos.
- Comprender y manipular los colores.
- Comprender y manipular las fuentes.
- Usar métodos de `Graphics` para dibujar líneas, rectángulos, rectángulos con esquinas redondeadas, rectángulos tridimensionales, óvalos, arcos y polígonos.
- Utilizar métodos de la clase `Graphics2D` de la `API Java2D` para dibujar líneas, rectángulos, rectángulos con esquinas redondeadas, elipses, arcos y rutas en general.
- Especificar las características `Paint` y `Stroke` de las figuras mostradas con `Graphics2D`.

Una imagen vale más que mil palabras.
Proverbio chino

Hay que tratar a la naturaleza en términos del cilindro, de la esfera, del cono, todo en perspectiva.
Paul Cezanne

Nada se vuelve real sino hasta que se experimenta; incluso un proverbio no será proverbio para usted, sino hasta que su vida lo haya ilustrado.
John Keats

Una imagen me muestra de un vistazo lo que requiere docenas de páginas de un libro para explicar.
Ivan Sergeyevich Turgenev

Plan general

12.1 Introducción
12.2 Contextos de gráficos y objetos de gráficos
12.3 Control de colores
12.4 Control de tipos de letra
12.5 Dibujo de líneas, rectángulos y óvalos
12.6 Dibujo de arcos
12.7 Dibujo de polígonos y polilíneas
12.8 La API Java2D
12.9 (Ejemplo práctico opcional) Acerca de los objetos: Diseño de interfaces con UML

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

12.1 Introducción

En este capítulo veremos varias de las herramientas de Java para dibujar figuras bidimensionales, controlar colores y fuentes. Uno de los atractivos iniciales de Java era su soporte para gráficos, el cual permitía a los programadores de Java mejorar la apariencia visual de sus applets (subprogramas) y sus aplicaciones. Ahora, Java contiene muchas más herramientas sofisticadas de dibujo como parte de la *API Java2D*. Comenzaremos este capítulo con una introducción a muchas de las herramientas de dibujo originales de Java. Después presentaremos varias de las nuevas y más poderosas herramientas de Java2D, como el control del estilo de líneas utilizadas para dibujar figuras y el control del relleno de las figuras con colores y patrones.

En la figura 12.1 se muestra una porción de la jerarquía de clases de Java que incluye varias de las clases de gráficos básicas y las clases e interfaces de la API Java2 que cubriremos en este capítulo. La clase `Color` contiene métodos y constantes para manipular los colores. La clase `Font` contiene métodos y constantes para manejar los tipos de letras. La clase `FontMetrics` contiene métodos para obtener información sobre los tipos de letras. La clase `Polygon` contiene métodos para crear polígonos. La clase `Graphics` contiene métodos para dibujar cadenas, líneas, rectángulos y demás figuras. La mitad inferior de la figura muestra varias clases e interfaces de la API Java2D. La clase `BasicStroke` ayuda a especificar las características de dibujo de las líneas. Las clases `GradientPaint` y `TexturePaint` ayudan a especificar las características para rellenar figuras con colores o patrones. Las clases `GeneralPath`, `Arc2D`, `Ellipse2D`, `Line2D`, `RectangleShape` y `RoundRectangle2D` representan varias figuras de Java2D.

Para empezar a dibujar en Java, primero debemos entender su *sistema de coordenadas* (figura 12.2), el cual es un esquema para identificar a cada uno de los posibles puntos en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente de la GUI (como un applet o una ventana) tiene las coordenadas (0,0). Un par de coordenadas está compuesto por una *coordenada x* (la *coordenada horizontal*) y una *coordenada y* (la *coordenada vertical*). La coordenada *x* es la distancia horizontal que se desplaza hacia la derecha, desde la esquina superior izquierda. La coordenada *y* es la distancia vertical que se desplaza hacia abajo, desde la esquina superior izquierda. El *eje x* describe cada una de las coordenadas horizontales, y el *eje y* describe cada una de las coordenadas verticales.

El texto y las figuras se muestran en la pantalla mediante la especificación de coordenadas. Las unidades de coordenada se miden en *píxeles*. Un píxel es la unidad de resolución más pequeña de la pantalla de un monitor de computadora.

La coordenada superior izquierda (0,0) de una ventana se encuentra *detrás* de la barra de título de la ventana. Por esta razón, las coordenadas de dibujo deben ajustarse para dibujar dentro de los bordes de la ventana. La clase `Container` (una superclase de todas las ventanas en Java) tiene el método `getInsets`, el cual devuelve un objeto `Insets` (paquete `java.awt`) para este fin. Un objeto `Insets` tiene cuatro miembros

http://librosysolucionarios.net

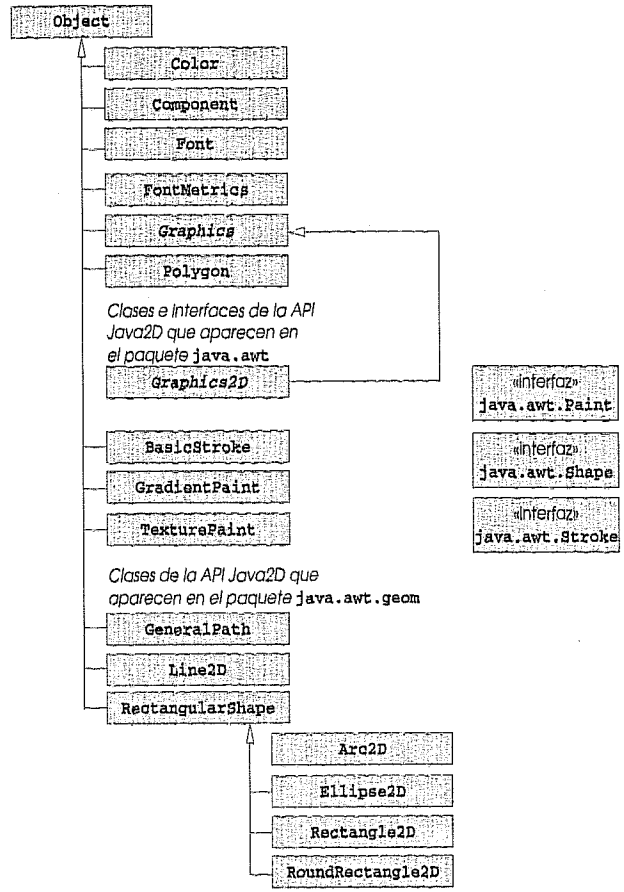


Figura 12.1 Clases e interfaces utilizadas en este capítulo, provenientes de las herramientas de gráficos originales de Java y de la API Java2D. (*Nota:* La clase `Object` aparece aquí porque es la superclase de la jerarquía de clases de Java.)

`public`: `top`, `bottom`, `left` y `right`, los cuales representan el número de píxeles desde cada extremo de la ventana hasta el área de dibujo para la ventana.



Tip de portabilidad 12.1

Existen distintos tipos de monitores de computadora con distintas resoluciones (es decir, la densidad de los píxeles varía). Esto puede hacer que los gráficos parezcan ser de distintos tamaños en distintos monitores, o en el mismo monitor con distintas configuraciones.

12.2 Contextos de gráficos y objetos de gráficos

Un *contexto de gráficos* en Java permite dibujar en la pantalla. Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla que representan texto y otros objetos gráficos (como líneas, elipses, rectángulos y otros polígonos). Los objetos `Graphics` contienen métodos para dibujar, manipular tipos de letra, manipular colores y demás. Todos los applets que hemos visto en el texto, que dibujan en la pantalla, han utiliza-

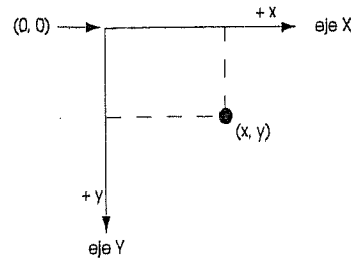


Figura 12.2 Sistema de coordenadas de Java. Las unidades se miden en pixeles.

do el objeto `Graphics g` (el argumento para el método `paint` del applet) para administrar el contexto de gráficos del applet. En este capítulo demostraremos cómo dibujar en las aplicaciones. Sin embargo, todas las técnicas que se mostrarán aquí pueden utilizarse en applets.

La clase `Graphics` es una clase `abstract` (es decir, no pueden instanciarse objetos `Graphics`). Esto contribuye a la portabilidad de Java. Como el dibujo se lleva a cabo de manera distinta en cada plataforma que soporta a Java, no puede haber una acción de implemento de las herramientas de dibujo en todos los sistemas. Por ejemplo, las herramientas de gráficos que permiten a una PC con Microsoft Windows dibujar un rectángulo, son distintas de las herramientas de gráficos que permiten a una estación de trabajo Linux dibujar un rectángulo; y ambas son distintas de las herramientas de gráficos que permiten a una Macintosh dibujar un rectángulo. Cuando Java se implementa en cada plataforma, se crea una subclase de `Graphics` que implementa las herramientas de dibujo. Esta implementación está oculta para nosotros por medio de la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar gráficos de una manera independiente de la plataforma.

La clase `Component` es la superclase para muchas de las clases en el paquete `java.awt`. (Hablaremos sobre esta clase en el capítulo 13.) El método `paint` de `Component` toma un objeto `Graphics` como argumento. El sistema pasa este objeto al método `paint` cuando se requiere una operación de pintar (`paint`) para un `Componente`. El encabezado del método `paint` es:

```
public void paint( Graphics g )
```

La referencia `g` tipo `Graphics` recibe una referencia a una instancia de la subclase específica del sistema que `Graphics` extiende. Tal vez a usted le parezca conocido el encabezado del método anterior; es el mismo que hemos estado utilizando en nuestras clases de applets. En realidad, la clase `Component` es una superclase indirecta de `JApplet`, la superclase de todos los applets de este libro. Muchas herramientas de la clase `JApplet` son heredadas de la clase `Component`.

El método `paint` raras veces es llamado directamente por el programador, ya que el dibujo de gráficos es un *proceso controlado por eventos*. Cuando se ejecuta un applet, el contenedor de applets llama al método `paint` (después de las llamadas a los métodos `init` y `start`). Para que `paint` sea llamado de nuevo, debe ocurrir un *evento* (como cubrir y descubrir el applet con otra ventana). De manera similar, cuando se muestra cualquier objeto `Component`, se hace una llamada al método `paint` de ese objeto.

Si el programador necesita llamar a `paint`, se hace una llamada al método `repaint` de `Component`. Este método solicita una llamada al método `update` de la clase `Component` lo más pronto posible, para borrar el fondo del objeto `Component`, quitando cualquier dibujo anterior, y después `update` llama directamente a `paint`. El método `repaint` es llamado con frecuencia por el programador, para forzar una operación `paint`. El método `repaint` no debe redefinirse, ya que realiza ciertas tareas dependientes del sistema. Los encabezados para `repaint` y `update` son:

```
public void repaint()
public void update( Graphics g )
```

El método `update` toma un objeto `Graphics` como argumento, el cual es suministrado automáticamente por el sistema cuando se hace la llamada a `update`.

http://librosysolucionarios.net

En este capítulo nos concentraremos en el método `paint`. En el siguiente capítulo hablaremos más detalladamente sobre la naturaleza controlada por eventos de los gráficos, y sobre los métodos `repaint` y `update`. También hablaremos sobre la clase `JComponent`: una superclase de muchos componentes de la GUI en el paquete `javax.swing`. Las subclases de `JComponent` generalmente pintan desde sus métodos `paintComponent`.

12.3 Control de colores

Los colores mejoran la apariencia de un programa y ayudan a transmitir su significado. Por ejemplo, un semáforo tiene tres luces de colores distintos: el rojo indica alto, el amarillo indica precaución y el verde indica que se puede pasar.

La clase `Color` declara los métodos y las constantes para manipular los colores en un programa de Java. Las constantes de colores previamente declaradas se sintetizan en la figura 12.3,[1] y varios métodos y constructores se sintetizan en la figura 12.4. Observe que dos de los métodos de la figura 12.4 son métodos de `Graphics` que son específicos para los colores.

Constante de color	Color	Valor RGB
`pubic final static Color ORANGE`	naranja	255, 200, 0
`public final static Color PINK`	rosa	255, 175, 175
`public final static Color CYAN`	cyan	0, 255, 255
`public final static Color MAGENTA`	magenta	255, 0, 255
`public final static Color YELLOW`	amarillo	255, 255, 0
`public final static Color BLACK`	negro	0, 0, 0
`public final static Color WHITE`	blanco	255, 255, 255
`public final static Color GRAY`	gris	128, 128, 128
`public final static Color LIGHT_GRAY`	gris claro	192, 192, 192
`public final static Color DARK_GRAY`	gris oscuro	64, 64, 64
`public final static Color RED`	rojo	255, 0, 0
`public final static Color GREEN`	verde	0, 255, 0
`public final static Color BLUE`	azul	0, 0, 255

Figura 12.3 Constantes de `Color` y sus valores RGB.

Método	Descripción
Constructores y métodos de `Color`	
`public Color( int r, int g, int b )`	Crea un color basado en los componentes rojo, verde y azul, expresados como enteros de 0 a 255.
`public Color( float r, float g, float b )`	Crea un color basado en los componentes rojo, verde y azul, expresados como valores de punto flotante de 0.0 a 1.0.

Figura 12.4 Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores. (Parte 1 de 2.)

1. En versiones anteriores de Java, las constantes de `Color` tenían nombres que consistían de letras mayúsculas y minúsculas. Aunque esos nombres todavía están disponibles, se prefiere el uso de los nombres que se muestran en la figura 12.3, ya que se conforman a las convenciones de nomenclatura para las constantes.

Método	Descripción
`public int getRed()`	Devuelve un valor entre 0 y 255, el cual representa el contenido rojo.
`public int getGreen()`	Devuelve un valor entre 0 y 255, el cual representa el contenido verde.
`public int getBlue()`	Devuelve un valor entre 0 y 255, el cual representa el contenido azul.
Métodos de `Graphics` *para manipular objetos* `Color`	
`public Color getColor()`	Devuelve un objeto `Color` que representa el color actual para el contexto de gráficos.
`public void setColor( Color c )`	Establece el color actual para dibujar con el contexto de gráficos.

Figura 12.4 Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores. (Parte 2 de 2.)

Todo color se crea a partir de un componente rojo, uno verde y otro azul. En conjunto, a estos componentes se les llama *valores RGB*. Los tres componentes RGB pueden ser enteros en el rango de 0 a 255, o pueden ser valores de punto flotante en el rango de 0.0 a 1.0. El primer componente RGB especifica la cantidad de rojo, el segundo, la cantidad de verde y el tercero, la cantidad de azul. Entre mayor sea el valor RGB, mayor será la cantidad de ese color en particular. Java permite al programador seleccionar de entre 256 × 256 × 256 (o aproximadamente 16.7 millones) colores. Sin embargo, no todas las computadoras son capaces de mostrar todos estos colores. Si éste es el caso, la computadora mostrará el color más cercano que pueda.

Los dos constructores de `Color` se muestran en la figura 12.4 (uno que toma tres argumentos `int` y otro que toma tres argumentos `float`, en donde cada argumento especifica la cantidad de rojo, verde y azul). Los valores `int` deben estar en el rango de 0 a 255 y los valores `float` deben estar en el rango de 0.0 a 1.0. El nuevo objeto `Color` tendrá las cantidades de rojo, azul y verde que se especifiquen. Los métodos `getRed`, `getGreen` y `getBlue` de `Color` devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente. El método `getColor` de `Graphics` devuelve un objeto `Color` que representa el color actual de dibujo. El método `setColor` de `Graphics` establece el color actual de dibujo.

La aplicación de la figura 12.5 demuestra varios métodos de la figura 12.4 al dibujar rectángulos rellenos y cadenas en varios colores distintos.

```
1  // Fig. 12.5: MostrarColores.java
2  // Demostración de los colores.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class MostrarColores extends JFrame {
7
8     // el constructor establece cadena de barra de título y dimensiones de la ventana
9     public MostrarColores()
10    {
11       super( "Uso de los colores" );
12
13       setSize( 400, 130 );
14       setVisible( true );
15    }
16
```

Figura 12.5 Cambio de colores para dibujar. (Parte 1 de 2.)

http://librosysolucionarios.net


```
17    // dibujar rectángulos y objetos String en distintos colores
18    public void paint( Graphics g )
19    {
20       // llamar al método paint de la superclase
21       super.paint( g );
22
23       // establecer nuevo color de dibujo utilizando enteros
24       g.setColor( new Color( 255, 0, 0 ) );
25       g.fillRect( 25, 25, 100, 20 );
26       g.drawString( "RGB actual: " + g.getColor(), 130, 40 );
27
28       // establecer nuevo color de dibujo utilizando valores float
29       g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
30       g.fillRect( 25, 50, 100, 20 );
31       g.drawString( "RGB actual: " + g.getColor(), 130, 65 );
32
33       // establecer nuevo color de dibujo utilizando objetos Color estáticos
34       g.setColor( Color.BLUE );
35       g.fillRect( 25, 75, 100, 20 );
36       g.drawString( "RGB actual: " + g.getColor(), 130, 90 );
37
38       // mostrar valores RGB individuales
39       Color color = Color.MAGENTA;
40       g.setColor( color );
41       g.fillRect( 25, 100, 100, 20 );
42       g.drawString( "Valores RGB: " + color.getRed() + ", " +
43          color.getGreen() + ", " + color.getBlue(), 130, 115 );
44
45    } // fin del método paint
46
47    // ejecutar la aplicación
48    public static void main( String args[] )
49    {
50       MostrarColores aplicacion = new MostrarColores();
51       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
52    }
53
54 } // fin de la clase MostrarColores
```

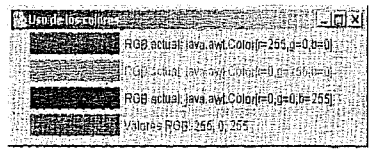


Figura 12.5 Cambio de colores para dibujar. (Parte 2 de 2.)

Cuando la aplicación empieza a ejecutarse, se hace una llamada al método `paint` de la clase `MostrarColores` (líneas 18 a 45) para pintar la ventana. En la línea 24 se utiliza el método `setColor` de `Graphics` para establecer el color actual de dibujo. El método `setColor` recibe un objeto `Color`. La expresión `new Color( 255, 0, 0 )` crea un nuevo objeto `Color` que representa rojo (valor `255` para rojo y `0` para los valores azul y verde). En la línea 25 se utiliza el método `fillRect` de `Graphics` para dibujar un rectángulo relleno con el color actual. El método `fillRect` recibe los mismos parámetros que el método `drawRect` (descrito en el capítulo 3). En la línea 26 se utiliza el método `drawString` de `Graphics` para dibujar un objeto `String` en el color actual. La expresión `g.getColor()` recupera el color actual del objeto `Graphics`.

El objeto `Color` devuelto se concatena con la cadena `"RGB actual:"`, lo que produce una llamada implícita al método `toString` de la clase `Color`. Observe que la representación `String` del objeto `Color` contiene el nombre de la clase y el paquete (`java.awt.Color`), además de los valores rojo, verde y azul.

En las líneas 29 a 31 y 34 a 36 se llevan a cabo, nuevamente, las mismas tareas. En la línea 29 se utiliza el constructor de `Color` con tres argumentos `float` para crear el color verde (`0.0f` para rojo, `1.0f` para verde y `0.0f` para azul). Observe la sintaxis de los valores. La letra `f` anexada a una literal de punto flotante indica que la literal debe tratarse como de tipo `float`. De manera predeterminada, las literales de punto flotante se tratan como de tipo `double`.

En la línea 34 se establece el color actual de dibujo a una de las constantes de `Color` previamente declaradas (`Color.BLUE`). Observe que `new` no se necesita para crear la constante. Las constantes de `Color` son `static`, por lo que se crean cuando la clase `Color` se carga en memoria, en tiempo de ejecución.

La instrucción de las líneas 42 y 43 demuestra el uso de los métodos `getRed`, `getGreen` y `getBlue` de `Color` en el objeto `Color.MAGENTA` previamente declarado.

Observación de ingeniería de software 12.1



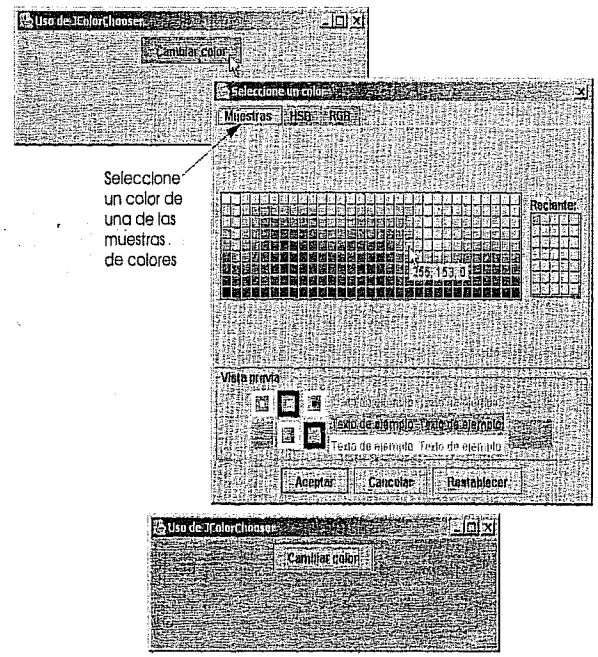
Para cambiar el color, debe crear un nuevo objeto Color (o utilizar una de las constantes de Color previamente declaradas). Los objetos Color no pueden modificarse (es decir, son inmutables).

El paquete `javax.swing` proporciona el componente de la GUI *`JColorChooser`* para permitir a los usuarios de aplicaciones seleccionar colores. En la figura 12.6 el lector podrá oprimir un botón para mostrar un cuadro de diálogo `JColorChooser`. Cuando seleccione un color y oprima el botón **Aceptar** del cuadro de diálogo, el color de fondo de la ventana de la aplicación cambiará de color.

```
1  // Fig. 12.6: MostrarColores2.java
2  // Seleccionar colores con JColorChooser.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class MostrarColores2 extends JFrame {
8     private JButton cambiarColorBoton;
9     private Color color = Color.LIGHT_GRAY;
10    private Container contenedor;
11
12    // configurar la GUI
13    public MostrarColores2()
14    {
15       super( "Uso de JColorChooser" );
16
17       contenedor = getContentPane();
18       contenedor.setLayout( new FlowLayout() );
19
20       // configurar cambiarColorBoton y registrar su manejador de eventos
21       cambiarColorBoton = new JButton( "Cambiar color" );
22       cambiarColorBoton.addActionListener(
23
24          new ActionListener() {  // clase interna anónima
25
26             // mostrar JColorChooser cuando el usuario haga clic en el botón
27             public void actionPerformed( ActionEvent evento )
28             {
29                color = JColorChooser.showDialog(
30                   MostrarColores2.this, "Seleccione un color", color );
31
```



```
32                // establecer color predeterminado, si no se devuelve un color
33                if ( color == null )
34                   color = Color.LIGHT_GRAY;
35
36                // cambiar color de fondo del panel de contenido
37                contenedor.setBackground( color );
38             }
39
40          } // fin de la clase interna anónima
41
42       ); // fin de la llamada a addActionListener
43
44       contenedor.add( cambiarColorBoton );
45
46       setSize( 400, 130 );
47       setVisible( true );
48
49    } // fin del constructor de MostrarColores2
50
51    // ejecutar la aplicación
52    public static void main( String args[] )
53    {
54       MostrarColores2 aplicacion = new MostrarColores2();
55       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
56    }
57
58 } // fin de la clase MostrarColores2
```



http://librosysolucionarios.net

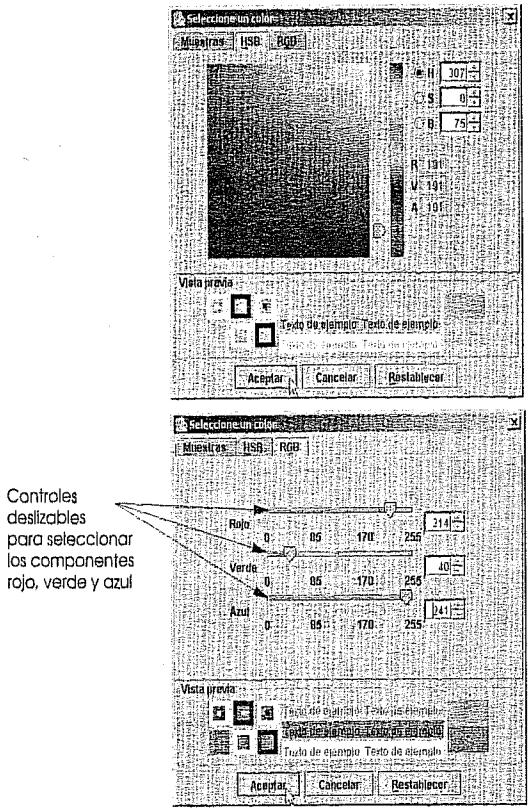


Figura 12.7 Las fichas **HSB** y **RGB** del cuadro de diálogo `JColorChooser`.

En las líneas 29 y 30 de `actionPerformed` se utiliza el método estático *`showDialog`* de `JColorChooser` para mostrar el cuadro de diálogo del seleccionador de colores. Este método devuelve el objeto `Color` seleccionado, o si el usuario oprime **Cancelar** o cierra el cuadro de diálogo sin oprimir **Aceptar**. Este método toma tres argumentos: una referencia a su objeto `Component` padre, un objeto `String` a mostrar en la barra de título del cuadro de diálogo y el `Color` inicial seleccionado para el cuadro de diálogo. El componente padre es una referencia a la ventana desde la que se muestra el cuadro de diálogo. Este cuadro de diálogo estará centrado en el componente padre. Si el padre es `null`, entonces el cuadro de diálogo se centra en la pantalla. Mientras el cuadro de diálogo para seleccionar colores se encuentre en la pantalla, el usuario no podrá interactuar con el componente padre. A este tipo de cuadro de diálogo se le conoce como *cuadro de diálogo modal* (el cual se describirá en el capítulo 14). Observe la sintaxis especial `MostrarColores2.this` que se utiliza en la línea 30. Recuerde que en la sección 10.9 vimos que puede tener acceso a la referencia `this` al objeto de la clase externa si califica a `this` con el nombre de la clase externa y un punto (.).

Una vez que el usuario selecciona un color, en las líneas 33 y 34 se determina si `color` es `null`, y de ser así `color` se establece en el valor predeterminado `Color.LIGHT_GRAY`. En la línea 37 se utiliza el método `setBackground` para cambiar el color de fondo del panel de contenido (representado por `contenedor` en este programa). El método `setBackground` es uno de los muchos métodos de la clase `Component` que pueden utilizarse en la mayoría de los componentes de la GUI.

La segunda captura de pantalla de la figura 12.6 muestra el cuadro de diálogo `JColorChooser` predeterminado, que permite al usuario seleccionar un color de una variedad de *muestras de colores*. Observe que en

realidad hay tres fichas en la parte superior del cuadro de diálogo: **Muestras**, **HSB** y **RGB**. Estas fichas representan tres distintas formas de seleccionar un color. La ficha **HSB** le permite seleccionar un color con base en *matiz (hue)*, *saturación (saturation)* y *brillo (brightness)*. La ficha **RGB** le permite seleccionar un color mediante el uso de controles deslizables para seleccionar los componentes rojo, verde y azul del color. Las fichas **HSB** y **RGB** se muestran en la figura 12.7.

12.4 Control de tipos de letra

En esta sección presentaremos los métodos y constantes para controlar los tipos de letras. La mayoría de los métodos y constantes de tipos de letra son parte de la clase `Font`. Algunos métodos de la clase `Font` y la clase `Graphics` se sintetizan en la figura 12.8.

Método o constante	Descripción
Constantes, constructores y métodos de `Font` para dibujar polígonos	
`public final static int PLAIN`	Constante que representa un estilo de tipo de letra simple.
`public final static int BOLD`	Constante que representa un estilo de tipo de letra en negritas.
`public final static int ITALIC`	Constante que representa un estilo de tipo de letra en cursivas.
`public Font( String nombre, int estilo, int tamaño )`	Crea un objeto `Font` con el nombre de tipo de letra, estilo y tamaño especificados.
`public int getStyle()`	Devuelve un valor entero que indica el estilo actual de tipo de letra.
`public int getSize()`	Devuelve un valor entero que indica el tamaño actual del tipo de letra.
`public String getName()`	Devuelve el nombre actual del tipo de letra, como una cadena.
`public String getFamily()`	Devuelve el nombre de la familia del tipo de letra, como una cadena.
`public boolean isPlain()`	Comprueba si un tipo de letra tiene estilo simple. Devuelve `true` si el tipo de letra es simple.
`public boolean isBold()`	Comprueba si un tipo de letra tiene estilo en negritas. Devuelve `true` si el tipo de letra está en negritas.
`public boolean isItalic()`	Comprueba si un tipo de letra tiene estilo en cursivas. Devuelve `true` si el tipo de letra está en cursivas.
Métodos de `Graphics` para manipular objetos `Font`	
`public Font getFont()`	Devuelve la referencia a un objeto `Font` que representa el tipo de letra actual.
`public void setFont( Font f )`	Establece el tipo de letra actual al nombre del tipo de letra, estilo y tamaño especificados por la referencia `f` al objeto `Font`.

Figura 12.8 Métodos y constantes relacionadas con la clase `Font`.

http://librosysolucionarios.net

El constructor de la clase **Font** toma tres argumentos: el *nombre del tipo de letra*, *su estilo* y *su tamaño*. El nombre del tipo de letra es cualquier tipo de letra soportado por el sistema en el que se esté ejecutando el programa, como los tipos de letra estándar de Java **Monospaced**, **SansSerif** y **Serif**. El estilo de tipo de letra es **Font.PLAIN** (simple), **Font.ITALIC** (cursivas) o **Font.BOLD** (negritas); cada uno es un campo **static** de la clase **Font**. Los estilos de los tipos de letra pueden usarse combinados (por ejemplo, **Font.ITALIC + Font.BOLD**). El tamaño del tipo de letra se mide en puntos. Un *punto* es 1/72 de una pulgada. El método **setFont** de **Graphics** establece el tipo de letra a dibujar en ese momento (el tipo de letra en el cual se mostrará el texto) en base a su argumento **Font**.



Tip de portabilidad 12.2

El número de tipos de letra varía enormemente entre sistemas. Java garantiza que los tipos de letra **Serif**, **Monospaced**, **SansSerif**, **Dialog** *y* **DialogInput** *estarán siempre disponibles.*



Error común de programación 12.1

Especificar un tipo de letra que no está disponible en un sistema es un error lógico. Java sustituirá el tipo de letra predeterminado de ese sistema (generalmente **Serif**).

En la figura 12.9 se muestra texto en cuatro tipos de letra distintos, con cada tipo de letra en diferente tamaño. El programa utiliza el constructor de **Font** para inicializar objetos **Font** (en las líneas 24, 28, 32 y 37) que se pasan al método **setFont** de **Graphics** para cambiar el tipo de letra para dibujar. Cada llamada al constructor de **Font** pasa un nombre de tipo de letra (**Serif**, **Monospaced**, o **SansSerif**) como una cadena, un estilo de tipo de letra (**Font.PLAIN**, **Font.ITALIC** o **Font.BOLD**) y un tamaño de tipo de letra. Una vez que se invoca el método **setFont** de **Graphics**, todo el texto que se muestre después de la llamada aparecerá en el nuevo tipo de letra hasta que éste se modifique. Observe que en la línea 36 se cambia el color de dibujo a rojo, por lo que la siguiente cadena que se muestra aparece en color rojo.

```
1  // Fig. 12.9: TiposDeLetra.java
2  // Uso de los tipos de letra.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class TiposDeLetra extends JFrame {
7
8     // establecer barra de título y dimensiones de la ventana
9     public TiposDeLetra()
10    {
11       super( "Uso de los tipos de letra" );
12
13       setSize( 450, 125 );
14       setVisible( true );
15    }
16
17    // mostrar cadenas en distintos tipos de letra y colores
18    public void paint( Graphics g )
19    {
20       // llamar al método paint de la superclase
21       super.paint( g );
22
23       // cambiar tipo de letra a Serif (Times), negrita, 12pts y dibujar una cadena
24       g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
25       g.drawString( "Serif 12 puntos, negrita.", 20, 50 );
26
27       // cambiar tipo de letra a Monospaced (Courier), cursiva, 24pts y dibujar una cadena
28       g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
```

Figura 12.9 El método **setFont** de **Graphics** cambia el tipo de letra para dibujar. (Parte 1 de 2.)


```
29       g.drawString( "Monospaced 24 puntos, cursiva.", 20, 70 );
30
31       // cambiar tipo de letra a SansSerif (Helvetica), simple, 14pts y dibujar una cadena
32       g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
33       g.drawString( "SansSerif 14 puntos, simple.", 20, 90 );
34
35       // cambiar tipo de letra a Serif (Times), negrita/cursiva, 18pts y dibujar una cadena
36       g.setColor( Color.RED );
37       g.setFont( new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
38       g.drawString( g.getFont().getName() + " " + g.getFont().getSize() +
39          " puntos, negrita cursiva.", 20, 110 );
40
41    } // fin del método paint
42
43    // ejecutar la aplicación
44    public static void main( String args[] )
45    {
46       TiposDeLetra aplicacion = new TiposDeLetra();
47       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
48    }
49
50 } // fin de la clase TiposDeLetra
```

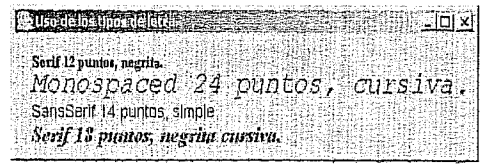


Figura 12.9 El método **setFont** de **Graphics** cambia el tipo de letra para dibujar (Parte 2 de 2.)



Observación de ingeniería de software 12.2

Para cambiar el tipo de letra, debe crear un nuevo objeto **Font**. *Los objetos* **Font** *son inmutables (*__Font__ *no tiene métodos* establecer *para modificar las características del tipo de letra actual).*

Métrica de los tipos de letra

En ocasiones, es necesario obtener información acerca del tipo de letra actual para dibujar, como el nombre, el estilo y el tamaño del tipo de letra. En la figura 12.8 se sintetizan varios métodos de **Font** que se utilizan para obtener información sobre el tipo de letra. El método **getStyle** devuelve un valor entero que representa el estilo actual. El valor entero devuelto puede ser **Font.PLAIN**, **Font.ITALIC**, **Font.BOLD** o la combinación de **Font.ITALIC** y **Font.BOLD**.

El método **getSize** devuelve el tamaño del tipo de letra en puntos. El método **getName** devuelve el nombre del tipo de letra actual como una cadena. El método **getFamily** devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de la plataforma.

También hay métodos de **Font** disponibles para probar el estilo del tipo de letra actual, los cuales se sintetizan en la figura 12.8. Los métodos **isPlain**, **isBold** e **isItalic** devuelven **true** si el estilo del tipo de letra actual es simple, negrita o cursiva, respectivamente.

A menudo, es necesario conocer información precisa acerca de la métrica de un tipo de letra, como la *altura*, el *descendente* (la distancia entre la base de la línea y el punto inferior del tipo de letra) y el *interlineado* (a veces llamado entre el descendente de una línea de texto y el ascendente de la línea de texto que está arriba; es decir, el espaciamiento entre líneas). En la figura 12.10 se muestran algunos elementos de la *métrica de los*

http://librosysolucionarios.net

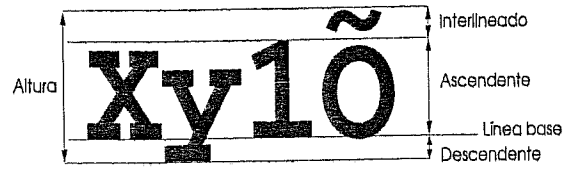


Figura 12.10 Métrica de los tipos de letra.

Método	Descripción
Métodos de **FontMetrics**	
`public int getAscent()`	Devuelve un valor que representa el ascendente de un tipo de letra, en puntos.
`public int getDescent()`	Devuelve un valor que representa el descendente de un tipo de letra, en puntos.
`public int getLeading()`	Devuelve un valor que representa el interlineado de un tipo de letra, en puntos.
`public int getHeight()`	Devuelve un valor que representa la altura de un tipo de letra, en puntos.
Métodos de **Graphics** *para obtener la métrica de un tipo de letra*	
`public FontMetrics getFontMetrics()`	Devuelve el objeto **FontMetrics** del tipo de letra (objeto **Font**) actual para dibujar.
`public FontMetrics getFontMetrics( Font f )`	Devuelve el objeto **FontMetrics** para el argumento **Font** especificado.

Figura 12.11 Métodos de **FontMetrics** y **Graphics** para obtener la información métrica de un tipo de letra.

```
1  // Fig. 12.12: Metrica.java
2  // Métodos de FontMetrics y Graphics útiles para obtener métrica de tipos de letra.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class Metrica extends JFrame {
7
8     // establecer cadena de la barra de título y dimensiones de la ventana
9     public Metrica()
10    {
11       super( "Demostración de FontMetrics" );
12
13       setSize( 540, 210 );
14       setVisible( true );
15    }
16
17    // mostrar métrica del tipo de letra
18    public void paint( Graphics g )
```

Figura 12.12 Métricas de los tipos de letra. (Parte 1 de 2.)

http://librosysolucionarios.net


```
19    {
20       super.paint( g );  // llamar al método paint de la superclase
21
22       g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
23       FontMetrics metrica = g.getFontMetrics();
24       g.drawString( "Tipo de letra actual: " + g.getFont(), 10, 40 );
25       g.drawString( "Ascendente: " + metrica.getAscent(), 10, 55 );
26       g.drawString( "Descendente: " + metrica.getDescent(), 10, 70 );
27       g.drawString( "Altura: " + metrica.getHeight(), 10, 85 );
28       g.drawString( "Interlineado: " + metrica.getLeading(), 10, 100 );
29
30       Font font = new Font( "Serif", Font.ITALIC, 14 );
31       metrica = g.getFontMetrics( font );
32       g.setFont( font );
33       g.drawString( "Tipo de letra actual: " + font, 10, 130 );
34       g.drawString( "Ascendente: " + metrica.getAscent(), 10, 145 );
35       g.drawString( "Descendente: " + metrica.getDescent(), 10, 160 );
36       g.drawString( "Altura: " + metrica.getHeight(), 10, 175 );
37       g.drawString( "Interlineado: " + metrica.getLeading(), 10, 190 );
38
39    } // fin del método paint
40
41    // ejecutar la aplicación
42    public static void main( String args[] )
43    {
44       Metrica aplicacion = new Metrica();
45       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
46    }
47
48 } // fin de la clase Metrica
```

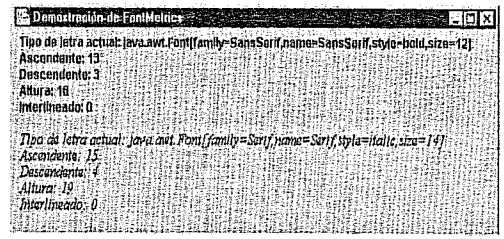


Figura 12.12 Métricas de los tipos de letra. (Parte 2 de 2.)

tipos de letras. Observe que la coordenada que se pasa a **drawString** corresponde a la esquina inferior izquierda de la línea base del tipo de letra.

La clase ***FontMetrics*** declara varios métodos para obtener información métrica de los tipos de letra. En la figura 12.11 se sintetizan estos métodos, junto con el método ***getFontMetrics*** de la clase **Graphics**. En la figura 12.12 se utilizan los métodos de la figura 12.11 para obtener la información métrica de dos tipos de letra.

En la línea 22 se crea y se establece el tipo de letra actual para dibujar en **SansSerif**, negrita, 12 puntos. En la línea 23 se utiliza el método **getFontMetrics** de la clase **Graphics** para obtener el objeto **FontMetrics** del tipo de letra actual. En la línea 24 se utiliza una llamada implícita al método **toString** de la clase **Font** para mostrar la representación de cadena del tipo de letra. En las líneas 25 a 28 se utilizan los métodos de **FontMetrics** para obtener el ascendente, descendente, altura e interlineado del tipo de letra.

En la línea 30 se crea un nuevo tipo de letra `Serif`, cursiva, 14 puntos. En la línea 31 se utiliza una segunda versión del método `getFontMetrics` de `Graphics`, la cual recibe un argumento `Font` y devuelve su correspondiente objeto `FontMetrics`. En las líneas 34 a 37 se obtiene el ascendente, descendente, altura e interlineado de ese tipo de letra. Observe que la métrica es ligeramente distinta para cada uno de los tipos de letra.

12.5 Dibujo de líneas, rectángulos y óvalos

En esta sección presentaremos varios métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. Los métodos y sus parámetros se sintetizan en la figura 12.13. Para cada método de dibujo que requiere un parámetro `anchura` y otro `altura`, sus valores deben ser números no negativos. De lo contrario, no se mostrará la figura. En la figura 12.14 demostramos cómo dibujar una variedad de líneas, rectángulos, rectángulos tridimensionales, rectángulos con esquinas redondeadas y óvalos.

Método	Descripción
`public void drawLine( int x1, int y1, int x2, int y2 )`	Dibuja una línea entre el punto (`x1`, `y1`) y el punto (`x2`, `y2`).
`public void drawRect( int x, int y, int anchura, int altura )`	Dibuja un rectángulo con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`).
`public void fillRect( int x, int y, int anchura, int altura )`	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`).
`public void clearRect( int x, int y, int anchura, int altura )`	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas, en el color de fondo actual. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`).
`public void drawRoundRect( int x, int y, int anchura, int altura, int anchuraArco, int alturaArco )`	Dibuja un rectángulo con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 12.15).
`public void fillRoundRect( int x, int y, int anchura, int altura, int anchuraArco, int alturaArco )`	Dibuja un rectángulo relleno con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 12.15).
`public void draw3DRect( int x, int y, int anchura, int altura, boolean b )`	Dibuja un rectángulo tridimensional en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`.
`public void fill3DRect( int x, int y, int anchura, int altura, boolean b )`	Dibuja un rectángulo tridimensional relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`.
`public void drawOval( int x, int y, int anchura, int altura )`	Dibuja un óvalo en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca los cuatro lados del rectángulo imaginario en el centro de cada uno de los lados (vea la figura 12.16).

Figura 12.13 Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. (Parte 1 de 2.)

Método	Descripción
`public void fillOval( int x, int y, int anchura, int altura )`	Dibuja un óvalo relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca los cuatro lados del rectángulo imaginario en el centro de cada uno de los lados (vea la figura 12.16).

Figura 12.13 Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. (Parte 2 de 2.)

```
1  // Fig. 12.14: LineasRectsOvalos.java
2  // Dibujo de líneas, rectángulos y óvalos.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class LineasRectsOvalos extends JFrame {
7
8     // establecer la cadena de la barra de título y dimensiones de la ventana
9     public LineasRectsOvalos()
10    {
11       super( "Dibujo de líneas, rectángulos y óvalos" );
12
13       setSize( 400, 165 );
14       setVisible( true );
15    }
16
17    // mostrar varias líneas, rectángulos y óvalos
18    public void paint( Graphics g )
19    {
20       super.paint( g );  // llamar al método paint de la superclase
21
22       g.setColor( Color.RED );
23       g.drawLine( 5, 30, 350, 30 );
24
25       g.setColor( Color.BLUE );
26       g.drawRect( 5, 40, 90, 55 );
27       g.fillRect( 100, 40, 90, 55 );
28
29       g.setColor( Color.CYAN );
30       g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
31       g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
32
33       g.setColor( Color.YELLOW );
34       g.draw3DRect( 5, 100, 90, 55, true );
35       g.fill3DRect( 100, 100, 90, 55, false );
36
37       g.setColor( Color.MAGENTA );
38       g.drawOval( 195, 100, 90, 55 );
39       g.fillOval( 290, 100, 90, 55 );
40
41    } // fin del método paint
42
43    // ejecutar la aplicación
44    public static void main( String args[] )
```

Figura 12.14 Dibujo de líneas, rectángulos y óvalos. (Parte 1 de 2.)

http://librosysolucionarios.net

```
45     {
46        LineasRectsOvalos aplicacion = new LineasRectsOvalos();
47        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
48     }
49
50  } // fin de la clase LineasRectsOvalos
```

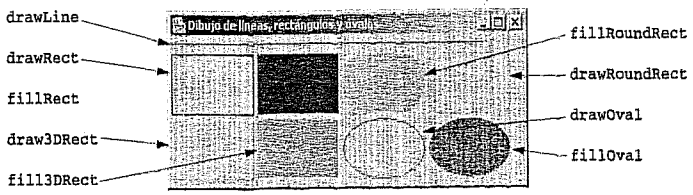


Figura 12.14 Dibujo de líneas, rectángulos y óvalos. (Parte 2 de 2.)

Los métodos `fillRoundRect` (línea 30) y `drawRoundRect` (línea 31) dibujan rectángulos con esquinas redondeadas. Sus primeros dos argumentos especifican las coordenadas de la esquina superior izquierda del *rectángulo delimitador* (el área en la que se va a dibujar el rectángulo redondeado). Observe que las coordenadas de la esquina superior izquierda no son el borde del rectángulo redondeado, sino las coordenadas en donde se encontraría el borde si el rectángulo tuviera esquinas cuadradas. Los argumentos tercero y cuarto especifican la `anchura` y `altura` del rectángulo. Sus últimos dos argumentos (`anchuraArco` y `alturaArco`) determinan los diámetros vertical y horizontal de los arcos utilizados para representar las esquinas.

En la figura 12.15 se muestran la anchura y altura del arco, junto con la anchura y la altura de un rectángulo redondeado. Si se utiliza el mismo valor para `anchuraArco` y `alturaArco` se produce un cuarto de círculo en cada esquina. Cuando `anchura`, `altura`, `anchuraArco` y `alturaArco` tienen los mismos valores, el resultado es un círculo. Si los valores para `anchura` y `altura` son los mismos, y los valores de `anchuraArco` y `alturaArco` son 0, el resultado es un cuadrado.

Los métodos `draw3DRect` (línea 34 de la figura 12.14) y `fill3DRect` (línea 35) toman los mismos argumentos. Los primeros dos argumentos especifican la esquina superior izquierda del rectángulo. Los siguientes dos argumentos especifican la `anchura` y `altura` del rectángulo, respectivamente. El último argumento determina si el rectángulo está *con relieve* (`true`) o *sin relieve* (`false`). El efecto tridimensional de `draw3DRect` aparece como dos bordes del rectángulo en el color original y dos bordes en un color ligeramente más oscuro. El efecto tridimensional de `fill3DRect` aparece como dos bordes del rectángulo en el color del dibujo original y los otros dos bordes y el relleno en un color ligeramente más oscuro. Los rectángulos con relieve tienen los bordes de color original del dibujo en las partes superior e izquierda del rectángulo. Los rectángulos sin relieve tienen los bordes de color original del dibujo en las partes inferior y derecha del rectángulo. El efecto tridimensional es difícil de ver en ciertos colores.

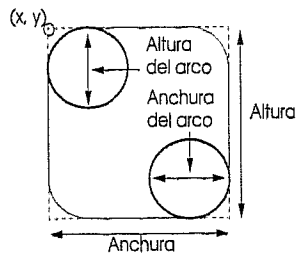


Figura 12.15 Anchura y altura del arco para los rectángulos redondeados.

Los métodos `drawOval` y `fillOval` (líneas 38 y 39) toman los mismos cuatro argumentos. Los primeros dos argumentos especifican la coordenada superior izquierda del rectángulo delimitador que contiene el óvalo. Los últimos dos argumentos especifican la `anchura` y la `altura` del rectángulo delimitador, respectivamente. En la figura 12.16 se muestra un óvalo delimitado por un rectángulo. Observe que el óvalo toca el centro de los cuatro lados del rectángulo delimitador. (El rectángulo delimitador no se muestra en la pantalla.)

12.6 Dibujo de arcos

Un *arco* se dibuja como una porción de un óvalo. Los ángulos de los arcos se miden en grados. Los arcos se *extienden* desde un *ángulo inicial*, en base al número de grados especificados por el *ángulo del arco*. El ángulo inicial indica, en grados, en dónde empieza el arco. El ángulo del arco especifica el número total de grados hasta los que se va a extender el arco. En la figura 12.17 se muestran dos arcos. El conjunto izquierdo de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente 110 grados. Los arcos que se extienden en dirección en contra de las manecillas del reloj se miden en *grados positivos*. El conjunto derecho de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente −110 grados. Los arcos que se extienden en dirección a favor de las manecillas del reloj se miden en *grados negativos*. Observe los cuadros punteados alrededor de los arcos en la figura 12.17. Cuando dibujamos un arco, debemos especificar un rectángulo delimitador para un óvalo. El arco se extenderá a lo largo de una parte del óvalo. Los métodos `drawArc` y `fillArc` de `Graphics` para dibujar arcos se sintetizan en la figura 12.18.

En la figura 12.19 se demuestra el uso de los métodos para arcos de la figura 12.18. El programa dibuja seis arcos (tres sin rellenar y tres rellenos). Para ilustrar el rectángulo delimitador que ayuda a determinar en dónde aparece el arco, los primeros tres arcos se muestran dentro de un rectángulo amarillo que tiene los mismos argumentos `x`, `y`, `anchura` y `altura` que los arcos.

12.7 Dibujo de polígonos y polilíneas

Los *polígonos* son figuras cerradas de varios lados, compuestas por segmentos de línea recta. Las *polilíneas* son una secuencia de puntos conectados. En la figura 12.20 describimos los métodos de `Graphics` para dibujar polí-

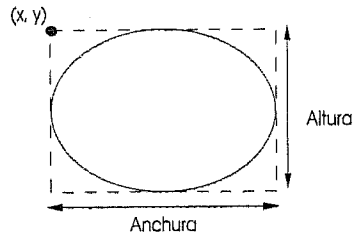


Figura 12.16 Un óvalo delimitado por un rectángulo.

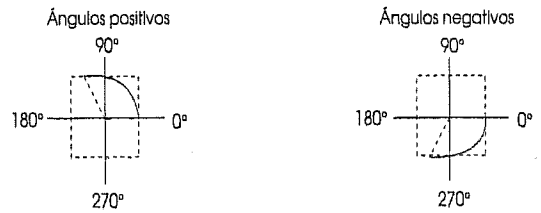


Figura 12.17 Ángulos positivos y negativos de un arco.

http://librosysolucionarios.net

Método	Descripción
`public void drawArc( int x, int y, int anchura, int altura, int anguloInicial, int anguloArco )`	Dibuja un arco relativo a las coordenadas (*x*, *y*) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.
`public void fillArc( int x, int y, int anchura, int altura, int anguloInicial, int anguloArco )`	Dibuja un arco relleno (es decir, un sector) relativo a las coordenadas (*x*, *y*) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.

Figura 12.18 Métodos de `Graphics` para dibujar arcos.

```
1  // Fig. 12.19: DibujarArcos.java
2  // Dibujo de arcos.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class DibujarArcos extends JFrame {
7
8     // establecer cadena de la barra de título y dimensiones de la ventana
9     public DibujarArcos()
10    {
11       super( "Dibujo de arcos" );
12
13       setSize( 300, 170 );
14       setVisible( true );
15    }
16
17    // dibujar rectángulos y arcos
18    public void paint( Graphics g )
19    {
20       super.paint( g );  // llamar al método paint de la superclase
21
22       // empezar en 0 y extenderse hasta 360 grados
23       g.setColor( Color.YELLOW );
24       g.drawRect( 15, 35, 80, 80 );
25       g.setColor( Color.BLACK );
26       g.drawArc( 15, 35, 80, 80, 0, 360 );
27
28       // empezar en 0 y extenderse hasta 110 grados
29       g.setColor( Color.YELLOW );
30       g.drawRect( 100, 35, 80, 80 );
31       g.setColor( Color.BLACK );
32       g.drawArc( 100, 35, 80, 80, 0, 110 );
33
34       // empezar en 0 y extenderse hasta -270 grados
35       g.setColor( Color.YELLOW );
```

Figura 12.19 Arcos mostrados con `drawArc` y `fillArc`. (Parte 1 de 2.)

http://librosysolucionarios.net


```
36       g.drawRect( 185, 35, 80, 80 );
37       g.setColor( Color.BLACK );
38       g.drawArc( 185, 35, 80, 80, 0, -270 );
39
40       // empezar en 0 y extenderse hasta 360 grados
41       g.fillArc( 15, 120, 80, 40, 0, 360 );
42
43       // empezar en 270 y extenderse hasta -90 grados
44       g.fillArc( 100, 120, 80, 40, 270, -90 );
45
46       // empezar en 0 y extenderse hasta -270 grados
47       g.fillArc( 185, 120, 80, 40, 0, -270 );
48
49    } // fin del método paint
50
51    // ejecutar la aplicación
52    public static void main( String args[] )
53    {
54       DibujarArcos aplicacion = new DibujarArcos();
55       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
56    }
57
58 } // fin de la clase DibujarArcos
```

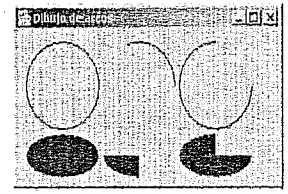


Figura 12.19 Arcos mostrados con `drawArc` y `fillArc`. (Parte 2 de 2.)

gonos y polilíneas. Observe que algunos métodos requieren un objeto *`Polygon`* (paquete `java.awt`). Los constructores de la clase `Polygon` se describen también en la figura 12.20. En la figura 12.21 se dibujan polígonos y polilíneas, utilizando los métodos y constructores de la figura 12.20.

En las líneas 22 y 23 se crean dos arreglos `int` y se utilizan para especificar los puntos del objeto `Polygon` llamado `poligono1`. La llamada al constructor de `Polygon` en la línea 24 recibe el arreglo `valoresX`, el cual contiene la coordenada *x* de cada punto; el arreglo `valoresY`, que contiene la coordenada *y*

Método	Descripción
Métodos de `Graphics` para dibujar polígonos	
`public void drawPolygon( int puntosX[], int puntosY[], int puntos )`	Dibuja un polígono. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un polígono cerrado. Si el último punto es distinto del primero, el polígono se cierra mediante una línea que conecte el último punto con el primero.

Figura 12.20 Métodos de `Graphics` para dibujar polígonos y métodos de la clase `Polygon`. (Parte 1 de 2.)

Método	Descripción
`public void drawPolyline( int puntosX[], int puntosY[], int puntos )`	Dibuja una secuencia de líneas conectadas. La coordenada *x* de cada punto se especifica en el arreglo **puntosX** y la coordenada *y* de cada punto se especifica en el arreglo **puntosY**. El último argumento especifica el número de **puntos**. Si el último punto es distinto del primero, la polilínea no se cierra.
`public void drawPolygon( Polygon p )`	Dibuja el polígono especificado.
`public void fillPolygon( int puntosX[], int puntosY[], int puntos )`	Dibuja un polígono relleno. La coordenada *x* de cada punto se especifica en el arreglo **puntosX** y la coordenada *y* de cada punto se especifica en el arreglo **puntosY**. El último argumento especifica el número de **puntos**. Este método dibuja un polígono cerrado. Si el último punto es distinto del primero, el polígono se cierra mediante una línea que conecte el último punto con el primero.
`public void fillPolygon( Polygon p )`	Dibuja el polígono relleno especificado. El polígono es cerrado.
Constructores y métodos de **Polygon**	
`public Polygon()`	Crea un nuevo objeto polígono. Este objeto no contiene ningún punto.
`public Polygon( int valoresX[], int valoresY[], int numeroDePuntos )`	Crea un nuevo objeto polígono. Este objeto tiene **numeroDePuntos** lados, en donde cada punto consiste de una coordenada *x* desde **valoresX**, y una coordenada *y* desde **valoresY**.
`public void addPoint( int x, int y )`	Agrega pares de coordenadas *x* y *y* al objeto **Polygon**.

Figura 12.20 Métodos de **Graphics** para dibujar polígonos y métodos de la clase **Polygon**. (Parte 2 de 2.)

```
1  // Fig. 12.21: DibujarPoligonos.java
2  // Dibujo de polígonos.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class DibujarPoligonos extends JFrame {
7
8     // establecer cadena de la barra de título y dimensiones de la ventana
9     public DibujarPoligonos()
10    {
11       super( "Dibujo de polígonos" );
12
13       setSize( 275, 230 );
14       setVisible( true );
15    }
16
17    // dibujar polígonos y polilíneas
18    public void paint( Graphics g )
19    {
20       super.paint( g );  // llamar al método paint de la superclase
```

Figura 12.21 Polígonos mostrados con **drawPolygon** y **fillPolygon**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
21
22       int valoresX[] = { 20, 40, 50, 30, 20, 15 };
23       int valoresY[] = { 50, 50, 60, 80, 80, 60 };
24       Polygon poligono1 = new Polygon( valoresX, valoresY, 6 );
25
26       g.drawPolygon( poligono1 );
27
28       int valoresX2[] = { 70, 90, 100, 80, 70, 65, 60 };
29       int valoresY2[] = { 100, 100, 110, 110, 130, 110, 90 };
30
31       g.drawPolyline( valoresX2, valoresY2, 7 );
32
33       int valoresX3[] = { 120, 140, 150, 190 };
34       int valoresY3[] = { 40, 70, 80, 60 };
35
36       g.fillPolygon( valoresX3, valoresY3, 4 );
37
38       Polygon poligono2 = new Polygon();
39       poligono2.addPoint( 165, 135 );
40       poligono2.addPoint( 175, 150 );
41       poligono2.addPoint( 270, 200 );
42       poligono2.addPoint( 200, 220 );
43       poligono2.addPoint( 130, 180 );
44
45       g.fillPolygon( poligono2 );
46
47    } // fin del método paint
48
49    // ejecutar la aplicación
50    public static void main( String args[] )
51    {
52       DibujarPoligonos aplicacion = new DibujarPoligonos();
53       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
54    }
55
56 } // fin de la clase DibujarPoligonos
```

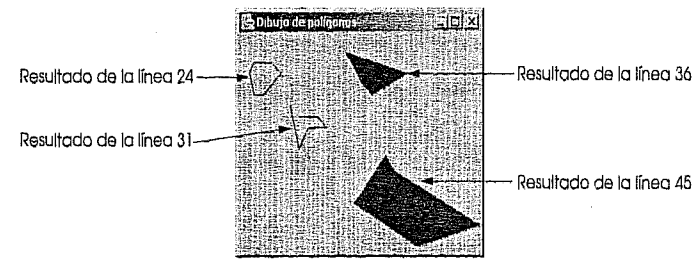


Figura 12.21 Polígonos mostrados con **drawPolygon** y **fillPolygon**. (Parte 2 de 2.)

de cada punto y el número 6 (el número de puntos en el polígono). En la línea 26 se muestra **poligono1** al pasarlo como argumento para el método **drawPolygon** de **Graphics**.

En las líneas 28 y 29 se crean dos arreglos **int** y se utilizan para especificar los puntos de una serie de líneas conectadas. El arreglo **valoresX2** contiene la coordenada *x* de cada punto y el arreglo **valoresY2**

contiene la coordenada y de cada punto. En la línea 31 se utiliza el método `drawPolyline` de `Graphics` para mostrar la serie de líneas conectadas que se especifican mediante los argumentos `valoresX2`, `valoresY2` y `7` (el número de puntos).

En las líneas 33 y 34 se crean dos arreglos `int` y se utilizan para especificar los puntos de un polígono. El arreglo `valoresX3` contiene la coordenada *x* de cada punto y el arreglo `valoresY3` contiene la coordenada y de cada punto. En la línea 36 se muestra un polígono al pasar al método `fillPolygon` de `Graphics` los dos arreglos (`valoresX3` y `valoresY3`) y el número de puntos a dibujar (`4`).



Error común de programación 12.2

Se lanzará una excepción `ArrayIndexOutOfBoundsException` si el número de puntos especificados en el tercer argumento del método `drawPolygon` o del método `fillPolygon` es mayor que el número de elementos en los arreglos de las coordenadas que especifican el polígono a mostrar.

En la línea 38 se crea el objeto `Polygon` llamado `poligono2`, sin puntos. En las líneas 39 a 43 se utiliza el método `addPoint` de `Polygon` para agregar pares de coordenadas *x* y *y* al objeto `Polygon`. En la línea 45 se muestra el objeto `Polygon` llamado `poligono2` al pasarlo al método `fillPolygon` de `Graphics`.

12.8 La API Java2D

La nueva *API Java2D* proporciona herramientas avanzadas para gráficos bidimensionales, para los programadores que requieren manipulaciones gráficas detalladas y complejas. La API incluye características para procesar arte lineal, texto e imágenes en los paquetes `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` y `java.awt.image.renderable`. Las herramientas de la API son muy extensas como para cubrirlas todas en este libro. Para ver las generalidades acerca de estas herramientas, consulte la demostración de Java2D (que se mostró en el capítulo 3) o visite la página Web `java.sun.com/products/java-media/2D/index.html`. En esta sección veremos las generalidades de varias herramientas de Java2D.

El dibujo con la API Java2 se logra mediante el uso de una instancia de `Graphics2D` (paquete `java.awt`), que es una subclase abstracta de la clase `Graphics`, por lo que tiene todas las herramientas para gráficos que se demostraron anteriormente en este capítulo. De hecho, el objeto en sí utilizado para dibujar en todos los métodos `paint` es una instancia de una subclase de `Graphics2D` que se pasa al método `paint` y se utiliza mediante el objeto `g` de la superclase `Graphics`. Para acceder a las herramientas de `Graphics2D`, debemos convertir la referencia `Graphics` que se pasa a `paint` en una referencia `Graphics2D`, mediante una instrucción como:

```
Graphics2D g2d = ( Graphics2D ) g;
```

Los siguientes dos ejemplos utilizan esta técnica.

Líneas, rectángulos, rectángulos redondeados, arcos y elipses

En el siguiente ejemplo se muestran varias figuras de Java2D del paquete `java.awt.geom`, incluyendo a `Line2D.Double`, `Rectangle2D.Double`, `RoundRectangle2D.Double`, `Arc2D.Double` y `Ellipse2D.Double`. Observe la sintaxis de cada uno de los nombres de las clases. Cada una de estas clases representa una figura con las dimensiones especificadas como valores de punto flotante con doble precisión. Hay una versión separada de cada figura, representada con valores de punto flotante con precisión simple (como `Ellipse2D.Float`). En cada caso, `Double` es una clase `static` anidada de la clase que se especifica a la izquierda del punto (por ejemplo, `Ellipse2D`). Para utilizar la clase `static` anidada, simplemente debemos calificar su nombre con el nombre de la clase externa.

En la figura 12.22 se muestra cómo dibujar figuras de Java2D y modificar sus características de dibujo, como cambiar el grosor de línea, rellenar figuras con patrones y dibujar líneas punteadas. Éstas son sólo algunas de las muchas herramientas que proporciona Java2D.

En la línea 24 se convierte la referencia `Graphics` recibida por `paint` a una referencia `Graphics2D`, y se asigna a `g2d` para permitir el acceso a las características de `Java2D`.

La primera figura que dibujamos es un óvalo relleno con colores que cambian gradualmente. En las líneas 27 y 28 se invoca el método `setPaint` de `Graphics2D` para establecer el objeto `Paint` que determina el


```
1  // Fig. 12.22: Figuras.java
2  // Demostración de algunas figuras de Java2D.
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.awt.image.*;
6  import javax.swing.*;
7
8  public class Figuras extends JFrame {
9
10    // establecer cadena de la barra de título y dimensiones de la ventana
11    public Figuras()
12    {
13       super( "Dibujo de figuras en 2D" );
14
15       setSize( 425, 160 );
16       setVisible( true );
17    }
18
19    // dibujar figuras con la API Java2D
20    public void paint( Graphics g )
21    {
22       super.paint( g );  // llamar al método paint de la superclase
23
24       Graphics2D g2d = ( Graphics2D ) g;  // convertir g a Graphics2D
25
26       // dibujar elipse 2D rellena con un degradado azul-amarillo
27       g2d.setPaint( new GradientPaint( 5, 30, Color.BLUE, 35, 100,
28          Color.YELLOW, true ) );
29       g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
30
31       // dibujar rectángulo 2D en rojo
32       g2d.setPaint( Color.RED );
33       g2d.setStroke( new BasicStroke( 10.0f ) );
34       g2d.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );
35
36       // dibujar rectángulo 2D redondeado con fondo tamponado
37       BufferedImage buffImage = new BufferedImage( 10, 10,
38          BufferedImage.TYPE_INT_RGB );
39
40       Graphics2D gg = buffImage.createGraphics();
41       gg.setColor( Color.YELLOW ); // dibujar en amarillo
42       gg.fillRect( 0, 0, 10, 10 ); // dibujar un rectángulo relleno
43       gg.setColor( Color.BLACK );  // dibujar en negro
44       gg.drawRect( 1, 1, 6, 6 );   // dibujar un rectángulo
45       gg.setColor( Color.BLUE );   // dibujar en azul
46       gg.fillRect( 1, 1, 3, 3 );   // dibujar un rectángulo relleno
47       gg.setColor( Color.RED );    // dibujar en rojo
48       gg.fillRect( 4, 4, 3, 3 );   // dibujar un rectángulo relleno
49
50       // pintar buffImage en el objeto JFrame
51       g2d.setPaint( new TexturePaint( buffImage,
52          new Rectangle( 10, 10 ) ) );
53       g2d.fill( new RoundRectangle2D.Double( 155, 30, 75, 100, 50, 50 ) );
54
55       // dibujar arco 2D en forma de pastel, en color blanco
```

Figura 12.22 Figuras de Java2D. (Parte 1 de 2.)

http://librosysolucionarios.net

```
56          g2d.setPaint( Color.WHITE );
57          g2d.setStroke( new BasicStroke( 6.0f ) );
58          g2d.draw( new Arc2D.Double( 240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
59
60          // dibujar líneas 2D en verde y amarillo
61          g2d.setPaint( Color.GREEN );
62          g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
63
64          float guiones[] = { 10 };
65
66          g2d.setPaint( Color.YELLOW );
67          g2d.setStroke( new BasicStroke( 4, BasicStroke.CAP_ROUND,
68             BasicStroke.JOIN_ROUND, 10, guiones, 0 ) );
69          g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
70
71       } // fin del método paint
72
73       // ejecutar la aplicación
74       public static void main( String args[] )
75       {
76          Figuras aplicacion = new Figuras();
77          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
78       }
79
80    } // fin de la clase Figuras
```

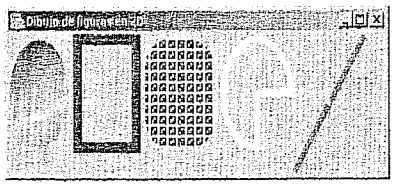


Figura 12.22 Figuras de Java2D. (Parte 2 de 2.)

color para la figura a mostrar. Un objeto `Paint` es un objeto de cualquier clase que implemente a la interfaz `java.awt.Paint`. El objeto `Paint` puede ser algo tan simple como uno de los objetos `Color` previamente declarados, los cuales se presentaron en la sección 12.3 (la clase `Color` implementa a `Paint`), o el objeto `Paint` puede ser una instancia de las clases ***`GradientPaint`***, ***`SystemColor`*** o ***`TexturePaint`*** de la API Java2. En este caso, utilizamos un objeto `GradientPaint`.

La clase `GradientPaint` ayuda a dibujar una figura en colores que cambian gradualmente (lo cual se conoce como *degradado*). El constructor de `GradientPaint` que se utiliza aquí requiere siete argumentos. Los primeros dos especifican la coordenada inicial del degradado. El tercer argumento especifica el `Color` inicial del degradado. Los argumentos cuarto y quinto especifican la coordenada final del degradado. El sexto especifica el `Color` final del degradado y el último especifica si el degradado es cíclico (`true`) o acíclico (`false`). Los dos conjuntos de coordenadas determinan la dirección del degradado. Como la segunda coordenada (*35, 100*) se encuentra hacia abajo y a la derecha de la primera coordenada (*5, 30*), el degradado va hacia abajo y a la derecha con cierto ángulo. Como este degradado es cíclico (`true`), el color empieza con azul, se convierte gradualmente en amarillo y luego regresa gradualmente a azul. Si el degradado es acíclico, el color cambia del primer color especificado (por ejemplo, azul) al segundo color (por ejemplo, amarillo).

En la línea 29 se utiliza el método ***`fill`*** de `Graphics2D` para dibujar un objeto ***`Shape`*** relleno (una instancia de cualquier clase que implemente a la interfaz `Shape` del paquete `java.awt`). En este caso mostramos una instancia de la clase `Ellipse2D.Double`. El constructor de `Ellipse2D.Double` recibe cuatro argumentos que especifican el rectángulo delimitador para mostrar la elipse.

A continuación, dibujamos un rectángulo rojo con un borde grueso. En la línea 32 se utiliza `setPaint` para establecer el objeto `Paint` en `Color.RED`. En la línea 33 se utiliza el método ***`setStroke`*** de `Graphics2D` para establecer las características del borde del rectángulo (o las líneas para cualquier otra figura). El método `setStroke` requiere como argumento un objeto ***`Stroke`***, el cual es una instancia de cualquier clase que implemente a la interfaz ***`Stroke`*** (paquete `java.awt`). En este caso, utilizamos una instancia de la clase ***`BasicStroke`***. Esta clase proporciona varios constructores para especificar la anchura de la línea, la manera en que ésta termina (lo cual se le conoce como *cofias*), la manera en que las líneas se unen entre sí (lo cual se le conoce como *uniones de línea*) y los atributos de los guiones de la línea (si es una línea punteada). El constructor aquí especifica que la línea debe tener una anchura de 10 píxeles.

En la línea 34 se utiliza el método ***`draw`*** de `Graphics2D` para dibujar un objeto `Shape`; en este caso, una instancia de la clase `Rectangle2D.Double`. El constructor de `Rectangle2D.Double` recibe cuatro argumentos que especifican las coordenadas *x* y *y* de la esquina superior izquierda, la anchura y la altura del rectángulo.

A continuación dibujamos un rectángulo redondeado, relleno con un patrón creado en un objeto ***`BufferedImage`*** (paquete `java.awt.image`). En las líneas 37 y 38 se crea el objeto `BufferedImage`. La clase `BufferedImage` puede usarse para producir imágenes en color y escala de grises. Este objeto `BufferedImage` en particular tiene una anchura y una altura de 10 píxeles . El tercer argumento del constructor, `BufferedImage.TYPE_INT_RGB`, indica que la imagen se almacena en color, utilizando el esquema de colores RGB.

Para crear el patrón de relleno para el rectángulo redondeado, debemos primero dibujar en el objeto `BufferedImage`. En la línea 40 se crea un objeto `Graphics2D` (con una llamada al método ***`createGraphics`*** de `BufferedImage`) que puede usarse para dibujar en el objeto `BufferedImage`. En las líneas 41 a 48 se utilizan los métodos `setColor`, `fillRect` y `drawRect` (descritos anteriormente en este capítulo) para crear el patrón.

En las líneas 51 y 52 se establece el objeto `Paint` en un nuevo objeto ***`TexturePaint`*** (paquete `java.awt`). Un objeto `TexturePaint` utiliza la imagen almacenada en su objeto `BufferedImage` asociado (el primer argumento del constructor) como la textura para rellenar una figura. El segundo argumento especifica el área `Rectangle` del objeto `BufferedImage` que se repetirá en toda la textura. En este caso, el objeto `Rectangle` es del mismo tamaño que el objeto `BufferedImage`. Sin embargo, puede utilizarse una porción más pequeña del objeto `BufferedImage`.

En la línea 53 se utiliza el método `fill` de `Graphics2D` para dibujar un objeto `Shape` relleno; en este caso, una instancia de la clase ***`RoundRectangle2D.Double`***. El constructor de la clase `RoundRectangle2D.Double` recibe seis argumentos que especifican las dimensiones del rectángulo, la anchura y la altura del arco utilizado para redondear las esquinas.

A continuación dibujamos un arco en forma de pastel, con una línea blanca gruesa. En la línea 56 se establece el objeto `Paint` en `Color.WHITE`. En la línea 57 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke` para una línea con 6 píxeles de anchura. En la línea 58 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, un `Arc2D.Double`. Los primeros cuatro argumentos del constructor de `Arc2D.Double` especifican las coordenadas *x* y *y* de la esquina superior izquierda, la anchura y la altura del rectángulo delimitador para el arco. El quinto argumento especifica el ángulo inicial. El sexto especifica el ángulo del arco; y el último especifica cómo se cierra el arco. La constante ***`Arc2D.PIE`*** indica que el arco se cierra dibujando dos líneas. Una línea va desde el punto inicial del arco hasta el centro del rectángulo delimitador, y otra va desde el centro del rectángulo delimitador hasta el punto final. La clase `Arc2D` proporciona otras dos constantes estáticas para especificar cómo se cierra el arco. La constante ***`Arc2D.CHORD`*** dibuja una línea que va desde el punto inicial hasta el punto final. La constante ***`Arc2D.OPEN`*** especifica que el arco no debe cerrarse.

Finalmente, dibujamos dos líneas utilizando objetos ***`Line2D`***: una sólida y una punteada. En la línea 61 se establece el objeto `Paint` en `Color.GREEN`. En la línea 62 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, una instancia de la clase `Line2D.Double`. Los argumentos del constructor de `Line2D.Double` especifican las coordenadas inicial y final de la línea.

En la línea 64 se declara un arreglo `float` de un elemento, el cual contiene el valor 10. Este arreglo debe utilizarse para describir los guiones en la línea punteada. En este caso, cada guión será de 10 píxeles de lar-

http://librosysolucionarios.net

go. Para crear guiones de diferentes longitudes en un patrón, simplemente debe proporcionar la longitud de cada guión como un elemento en el arreglo. En la línea 66 se establece el objeto **Paint** en **Color.YELLOW**. En las líneas 67 y 68 se establece el objeto **Stroke** en un nuevo objeto **BasicStroke**. La línea tendrá una anchura de 4 píxeles y extremos redondeados (**BasicStroke.CAP_ROUND**). Si las líneas se unen entre sí (como en un rectángulo en las esquinas), la unión de la líneas será redondeada (**BasicStroke.JOIN_ROUND**). El argumento **guiones** especifica las longitudes de los guiones de la línea. El último argumento indica el índice inicial en el arreglo **guiones** para el primer guión en el patrón. En la línea 69 se dibuja una línea con el objeto **Stroke** actual.

Rutas generales

A continuación presentaremos una ruta general: una figura compuesta de líneas rectas y curvas complejas. Una ruta general se representa con un objeto de la clase ***GeneralPath*** (paquete **java.awt.geom**). En la figura 12.23 se demuestra cómo dibujar una ruta general, en forma de una estrella de cinco puntas.

En las líneas 24 y 25 se declaran dos arreglos **int** que representan las coordenadas *x* y *y* de los puntos en la estrella. En la línea 28 se crea el objeto **GeneralPath** llamado **estrella**.

```
1  // Fig. 12.23: Figuras2.java
2  // Demostración de una ruta general.
3  import java.awt.*;
4  import java.awt.geom.*;
5  import javax.swing.*;
6
7  public class Figuras2 extends JFrame {
8
9     //establecer cadena de la barra de título, color de fondo y dimensiones de la ventana
10    public Figuras2()
11    {
12       super( "Dibujo de figuras 2D" );
13
14       getContentPane().setBackground( Color.WHITE );
15       setSize( 400, 400 );
16       setVisible( true );
17    }
18
19    // dibujar rutas generales
20    public void paint( Graphics g )
21    {
22       super.paint( g );  // llamar al método paint de la superclase
23
24       int puntosX[] = { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
25       int puntosY[] = { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
26
27       Graphics2D g2d = ( Graphics2D ) g;
28       GeneralPath estrella = new GeneralPath(); // crear objeto GeneralPath
29
30       // establecer la coordenada inicial de la ruta general
31       estrella.moveTo( puntosX[ 0 ], puntosY[ 0 ] );
32
33       // crear la estrella--esto no la dibuja
34       for ( int cuenta = 1; cuenta < puntosX.length; cuenta++ )
35          estrella.lineTo( puntosX[ cuenta ], puntosY[ cuenta ] );
36
```

Figura 12.23 Rutas generales de Java2D. (Parte 1 de 2.)

http://librosysolucionarios.net


```
37       estrella.closePath();  // cerrar la figura
38
39       g2d.translate( 200, 200 );  // trasladar el origen a (200, 200)
40
41       // girar alrededor del origen y dibujar estrellas en colores aleatorios
42       for ( int cuenta = 1; cuenta <= 20; cuenta++ ) {
43          g2d.rotate( Math.PI / 10.0 );  // girar el sistema de coordenadas
44
45          // establecer color de dibujo al azar
46          g2d.setColor( new Color( ( int ) ( Math.random() * 256 ),
47             ( int ) ( Math.random() * 256 ),
48             ( int ) ( Math.random() * 256 ) ) );
49
50          g2d.fill( estrella );  // dibujar estrella rellena
51       }
52
53    } // fin del método paint
54
55    // ejecutar la aplicación
56    public static void main( String args[] )
57    {
58       Figuras2 aplicacion = new Figuras2();
59       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
60    }
61
62 } // fin de la clase Figuras2
```

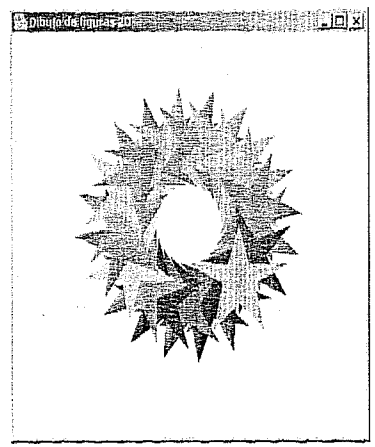


Figura 12.23 Rutas generales de Java2D. (Parte 2 de 2.)

En la línea 31 se utiliza el método ***moveTo*** de **GeneralPath** para especificar el primer punto en la **estrella**. La instrucción **for** de las líneas 34 y 35 utiliza el método ***lineTo*** de **GeneralPath** para dibujar una línea al siguiente punto en la **estrella**. Cada nueva llamada a **lineTo** dibuja una línea del punto anterior al punto actual. En la línea 37 se utiliza el método ***closePath*** de **GeneralPath** para dibujar una línea del último punto hasta el punto especificado en la última llamada a **moveTo**. Esto completa la ruta general.

En la línea 39 se utiliza el método ***translate*** de **Graphics2D** para desplazar el origen del dibujo hasta la ubicación (*200, 200*). Todas las operaciones de dibujo utilizarán ahora la ubicación (*200, 200*) como si fuera (*0, 0*).

La instrucción `for` de las líneas 42 a 51 dibujan la `estrella` 20 veces, girándola alrededor del nuevo punto del origen. En la línea 43 se utiliza el método `rotate` de `Graphics2D` para girar la siguiente figura a mostrar. El argumento especifica el ángulo de giro en radianes (con 360° = 2π radianes). En la línea 50 se utiliza el método `fill` de `Graphics2D` para dibujar una versión rellena de la `estrella`.

12.9 (Ejemplo práctico opcional) Acerca de los objetos: Diseño de interfaces con UML

En la sección 11.9 incorporamos el manejo de eventos en nuestra simulación, modificando el diagrama de colaboración que se encarga de la entrada y salida de pasajeros al elevador. Incluimos tanto el manejo de eventos como la herencia en ese diagrama. El `Elevador` informa a su `Puerta` de su llegada. Esta `Puerta` abre la `Puerta` del `Piso` al que llegó el `Elevador`, obteniendo su manejador a través de un objeto `Ubicacion` (el cual se incluyó en el evento de llegada), y potencialmente dos objetos `Persona` salen y entran al `Elevador` después de que ambas `Puertas` se abren. También hablamos sobre las interfaces de escucha. En esta sección representaremos a nuestra interfaz de escucha con UML.

Realizaciones

UML expresa la relación entre una clase y una interfaz a través de una *realización*. Una clase *realiza*, o implementa, los comportamientos de una interfaz. Un diagrama de clases puede mostrar una realización entre las clases y las interfaces. En la figura 12.24 se modela la realización entre la clase `Persona` y la interfaz `EscuchaPuerta`. La relación entre `Persona` y `EscuchaPuerta` parece ser similar a una generalización, con la excepción de que la flecha que expresa la relación es punteada, en vez de sólida. Observe que el compartimiento de en medio en la interfaz `EscuchaPuerta` está vacío, ya que las interfaces no contienen atributos. Por último, observe la palabra "InterfazJava" colocada entre los signos (« »), ubicados en el primer compartimiento de la interfaz `EscuchaPuerta`. Esta notación distingue a la interfaz `EscuchaPuerta` como una interfaz de Java en nuestro sistema. Los elementos que se colocan entre los signos « » se conocen como *estereotipos* en UML. Un estereotipo es una extensión de UML. En este caso, el estereotipo « InterfazJava » extiende a UML, de manera que podamos modelar las interfaces de Java.[2]

En la figura 12.25 se muestra la manera alternativa para representar la realización de la clase `persona` y la interfaz `EscuchaPuerta` en UML. La figura 12.25 es el diagrama con elementos omitidos de la figura 12.24. El círculo pequeño representa la interfaz, y la línea sólida representa la realización. Al ocultar sus operaciones, condensamos la interfaz, lo cual mejora su legibilidad; sin embargo, al hacer esto sacrificamos la información acerca de sus comportamientos. Al construir un diagrama con elementos omitidos, la práctica común es colocar la información relacionada con cualquier comportamiento en un diagrama separado; por ejemplo, colocamos la interfaz completa `EscuchaPuerta` en el diagrama de clases de la figura 12.28.

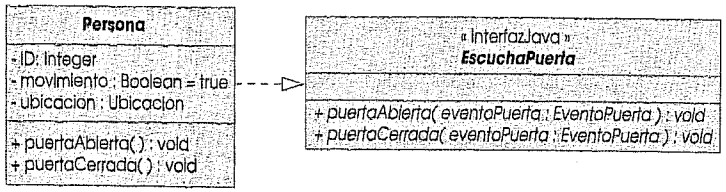


Figura 12.24 Diagrama de clases que modela a la clase `Persona` realizando la interfaz `EscuchaPuerta`.

2. De acuerdo con la especificación de UML versión 1.4, las interfaces no tienen atributos. Sin embargo, las interfaces de Java pueden tener atributos (en forma de campos `static final`). Mediante el Proceso de la comunidad Java (JCP, Java Community Process), la comunidad de Java está desarrollando una asignación de UML a Java que incluye soporte para « InterfazJava » (Solicitud de especificación Java 26). Para obtener más información sobre el JCP, visite el sitio `www.jcp.org`.

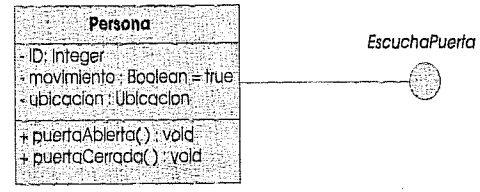


Figura 12.25 Diagrama de clases con elementos omitidos, que modela a la clase `Persona` realizando la interfaz `EscuchaPuerta`.

El proceso de aplicar ingeniería hacia delante de UML a código implementado en Java se beneficia de los diagramas de realización bien construidos. Al declarar cualquier clase, debe especificar la realización entre la clase y su interfaz (esa clase "implementará" a la interfaz y redefinirá los métodos de ésta). Por ejemplo, utilizamos la figura 12.24 para empezar a construir `Persona.java`. En la figura 12.26 se muestra la implementación de Java para la figura 12.24. En las líneas 6 a 8 y en las líneas 14 y 15 se declaran los atributos y operaciones de `Persona`, incluyendo aquellos requeridos por la interfaz `EscuchaPuerta`.

Cuando una `Puerta` se abre o cierra, esa `Puerta` invoca al método apropiado que está declarado en la interfaz `EscuchaPuerta`, pero solamente si la `Persona` está registrada con esa `Puerta` para recibir eventos tipo `EventoPuerta`. Finalmente, presentamos un modelo con elementos omitidos de las realizaciones en nuestro modelo del elevador, en la figura 12.27. El diagrama con elementos omitidos no tiene ningún método de interfaz (haciendo que el diagrama sea fácil de leer). Por lo tanto, modelamos las interfaces en la figura 12.28, la cual muestra todos los métodos de la interfaz. Modelaremos la realización de la interfaz `EscuchaMoverPersona` en la sección 14.13. Consulte estos diagramas cuando estudie la implementación de la simulación del elevador en los apéndices G, H e I.

De acuerdo con la figura 12.27, las clases `Puerta`, `Luz`, `Timbre` y `Boton` implementan la interfaz `EscuchaMoverElevador`. La clase `Elevador` implementa las interfaces `EscuchaBoton`, `EscuchaPuerta` y `EscuchaTimbre`. La clase `ConductoElevador` implementa las interfaces `EscuchaLuz`, `EscuchaBoton` y `EscuchaPuerta`. Por último, la clase `Persona` implementa la interfaz `EscuchaPuerta`. En el apéndice E analizaremos nuevamente la figura 12.27, cuando comencemos a codificar nuestro modelo.

En esta sección mostramos cómo representar interfaces y realizaciones con UML. También presentamos diagramas de clases que muestran las interfaces de escucha y sus realizaciones para nuestra simulación del elevador. En la sección 14.13 modelaremos cómo el usuario interactúa con nuestra simulación.

```
1  // Persona.java
2  // Generado a partir de la figura 11.24
3  public class Persona implements EscuchaPuerta {
4
5     // atributos
6     private int ID;
7     private boolean movimiento = true;
8     private Ubicacion ubicacion;
9
10    // constructor
11    public Persona() {}
12
13    // métodos de EscuchaPuerta
14    public void puertaAbierta( EventoPuerta eventoPuerta ) {}
15    public void puertaCerrada( EventoPuerta eventoPuerta ) {}
16 }
```

Figura 12.26 La clase `Persona` se genera a partir de la figura 12.24.

http://librosysolucionarios.net

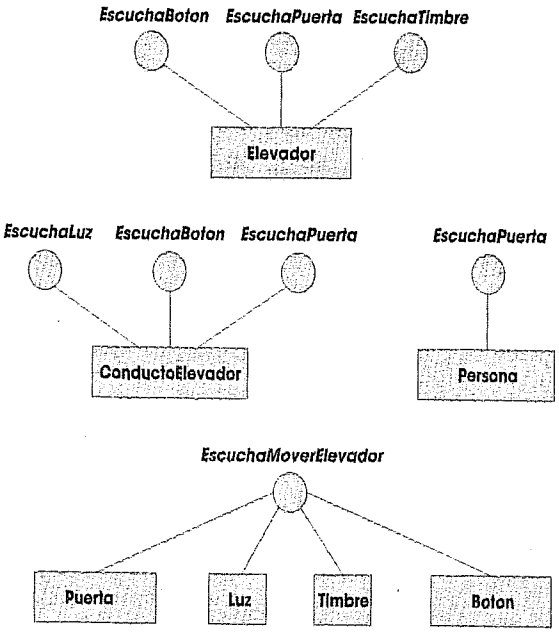


Figura 12.27 Diagrama de clases que modela las realizaciones en el modelo del elevador.

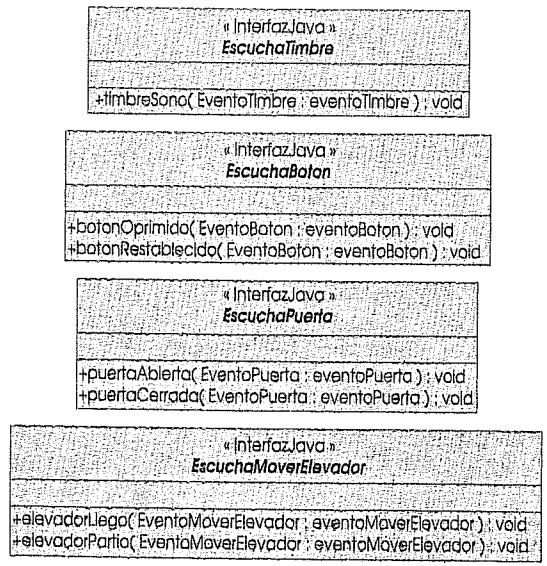


Figura 12.28 Diagrama de clases para las interfaces de escucha. (Parte 1 de 2.)

http://librosysolucionarios.net



Figura 12.28 Diagrama de clases para las interfaces de escucha. (Parte 2 de 2.)

RESUMEN

- Un sistema de coordenadas es un esquema para identificar todos los posibles puntos en la pantalla.
- La esquina superior izquierda de un componente de la GUI tiene las coordenadas *(0, 0)*. Un par de coordenadas está compuesto de una coordenada *x* (la coordenada horizontal) y una coordenada *y* (la coordenada vertical).
- Las unidades de coordenadas se miden en píxeles. Un píxel es la unidad de resolución más pequeña de un monitor de computadora.
- Un contexto de gráficos en Java permite dibujar en la pantalla. Un objeto `Graphics` administra un contexto de gráficos, controlando la manera en que se muestran los píxeles.
- Los objetos `Graphics` contienen métodos para dibujar, manipular tipos de letra y colores, etcétera.
- El método `paint` normalmente se llama en respuesta a un evento, como descubrir una ventana.
- El método `repaint` solicita una llamada al método `update` de `Component` lo más pronto posible para borrar el fondo del objeto `Component`, quitando cualquier dibujo anterior, y después `update` llama a `paint` directamente.
- La clase `Color` declara métodos y constantes para manipular colores en un programa de Java.
- Java utiliza colores RGB, en donde los componentes de color rojo, verde y azul son enteros en el rango de 0 a 255, o valores de punto flotante en el rango de 0.0 a 1.0.
- Los métodos `getRed`, `getGreen` y `getBlue` de `Color` devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul en un objeto `Color`. La clase `Color` proporciona 13 objetos `Color` previamente declarados.
- El método `getColor` de `Graphics` devuelve un objeto `Color` que representa el color actual para dibujar. El método `setColor` de `Graphics` establece el color actual para dibujar.
- Java proporciona la clase `JColorChooser` para mostrar un cuadro de diálogo para seleccionar colores. El método `showDialog` de `JColorChooser` muestra un cuadro de diálogo para seleccionar colores y devuelve el objeto `Color` seleccionado.
- El cuadro de diálogo `JColorChooser` predeterminado le permite seleccionar un color de una variedad de muestras de colores. La ficha **HSB** le permite seleccionar un color con base en el matiz, la saturación y el brillo. La ficha **RGB** le permite seleccionar un color utilizando controles deslizables para los componentes rojo, verde y azul del color.
- El método `setBackground` de `Component` cambia el color de fondo de un componente.
- El constructor de la clase `Font` toma tres argumentos: el nombre, estilo y tamaño del tipo de letra. El nombre del tipo de letra puede ser cualquier tipo de letra soportado por el sistema. El estilo del tipo de letra puede ser `Font.PLAIN`, `Font.ITALIC` o `Font.BOLD`. El tamaño del tipo de letra se mide en puntos.
- El método `setFont` de `Graphics` establece el tipo de letra para dibujar.
- La clase `FontMetrics` declara varios métodos para obtener la métrica de los tipos de letra. El método `getFontMetrics` de `Graphics` sin argumentos obtiene el objeto `FontMetrics` para el tipo de letra actual. El método `getFontMetrics` de `Graphics` con un argumento `Font` devuelve el objeto `FontMetrics` de ese objeto `Font`.
- Los métodos `draw3DRect` y `fill3DRect` toman cinco argumentos que especifican la esquina superior izquierda del rectángulo, la `anchura` y `altura` del rectángulo, y si el rectángulo va con relieve o sin él.

- Los métodos `drawRoundRect` y `fillRoundRect` dibujan rectángulos con esquinas redondeadas. Sus primeros dos argumentos especifican la esquina superior izquierda, los argumentos tercero y cuarto especifican la `anchura` y `altura`, y los últimos dos argumentos (`anchuraArco` y `alturaArco`) determinan los diámetros vertical y horizontal de los arcos utilizados para representar las esquinas.
- Los métodos `drawOval` y `fillOval` toman los mismos argumentos: la coordenada de la esquina superior izquierda y la `anchura` y `altura` del rectángulo delimitador que contiene al óvalo.
- Los arcos se extienden desde un ángulo inicial hasta el número de grados especificados por su ángulo de arco. El ángulo inicial especifica en dónde empieza el arco, y el ángulo del arco especifica el número total de grados hasta los que se extenderá el arco. Los arcos que se extienden en sentido contrario de las manecillas del reloj se miden en grados positivos, y los ángulos que se extienden en sentido a favor de las manecillas del reloj se miden en grados negativos.
- Los métodos `drawArc` y `fillArc` toman los mismos argumentos: la coordenada de la esquina superior izquierda, la `anchura` y `altura` del rectángulo delimitador que contiene al arco, y los argumentos `anguloInicial` y `arcoAngulo` que especifican la extensión del arco.
- Los polígonos son figuras cerradas de varios lados, compuestas de segmentos rectos. Las polilíneas son una secuencia de puntos conectados.
- Una versión del método `drawPolygon` de `Graphics` muestra a un objeto `Polygon`. Otra versión recibe un arreglo que contiene la coordenada *x* de cada punto, un arreglo que contiene la coordenada *y* de cada punto y el número de puntos en el polígono, y muestra el polígono correspondiente.
- El método `drawPolyline` de `Graphics` muestra una serie de líneas conectadas, especificadas por sus argumentos (un arreglo que contiene la coordenada *x* de cada punto, un arreglo que contiene la coordenada *y* de cada punto y el número de puntos).
- El método `addPoint` de `Polygon` agrega pares de coordenadas *x* y *y* a un objeto `Polygon`.
- La API Java2D proporciona herramientas para gráficos bidimensionales avanzados, para procesar arte lineal, texto e imágenes.
- Para acceder a las herramientas de `Graphics2D`, se debe aplicar una conversión descendente a la referencia `Graphics` que se pasa a `paint`, para convertirla en una referencia `Graphics2D`.
- El método `setPaint` de `Graphics2D` establece el objeto `Paint` que determina el color y la textura para la figura a mostrar. El objeto `Paint` puede ser un objeto `Color` o una instancia de las clases `GradientPaint`, `SystemColor` o `TexturePaint` de la API Java2.
- La clase `GradientPaint` dibuja una figura en un color que cambia gradualmente, conocido como degradado.
- El método `fill` de `Graphics2D` dibuja un objeto `Shape` relleno.
- El constructor de `Ellipse2D.Double` recibe cuatro argumentos, los cuales especifican el rectángulo delimitador para la elipse a mostrar.
- El método `setStroke` de `Graphics2D` establece las características de las líneas utilizadas para dibujar una figura.
- El método `draw` de `Graphics2D` dibuja un objeto `Shape`.
- El constructor de `Rectangle2D.Double` recibe cuatro argumentos que especifican las coordenadas *x* y *y* de la esquina superior izquierda, la anchura y la altura del rectángulo.
- La clase `BufferedImage` puede utilizarse para producir imágenes en color y escala de grises.
- Un objeto `TexturePaint` utiliza la imagen almacenada en su objeto `BufferedImage` asociado como la textura para rellenar una figura.
- El constructor de `RoundRectangle2D.Double` recibe seis argumentos que especifican las dimensiones del rectángulo, la anchura y la altura del arco utilizados para determinar la redondez de las esquinas.
- Los primeros cuatro argumentos del constructor de `Arc2D.Double` especifican las coordenadas *x* y *y* de la esquina superior izquierda, la anchura y la altura del rectángulo delimitador para el arco. El quinto argumento especifica el ángulo inicial. El sexto especifica el ángulo final; y el último especifica el tipo de arco (`Arc2D.PIE`, `Arc2D.CHORD` o `Arc2D.OPEN`).
- Los argumentos del constructor de `Line2D.Double` especifican las coordenadas inicial y final de la línea.
- Una ruta general es una figura que se construye a partir de líneas rectas y curvas complejas, las cuales se representan con un objeto de la clase `GeneralPath` (paquete `java.awt.geom`).
- El método `moveTo` de `GeneralPath` especifica el primer punto en una ruta general. El método `lineTo` de `GeneralPath` dibuja una línea hasta el siguiente punto en la ruta general. Cada nueva llamada a `lineTo` dibuja una línea del punto anterior al punto actual. El método `closePath` de `GeneralPath` dibuja una línea desde el último punto hasta el punto especificado en la última llamada a `moveTo`.
- El método `translate` de `Graphics2D` desplaza el origen del dibujo hacia una nueva ubicación. Todas las operaciones de dibujo utilizarán ahora esa ubicación como si fuera *(0, 0)*. El método `rotate` de `Graphics2D` permite girar la siguiente figura a mostrar. Su argumento especifica el ángulo de giro en radianes (en donde $360° = 2\pi$ radianes).

TERMINOLOGÍA

`addPoint`, método de `Polygon`
altura del arco
anchura del arco
`Arc2D.Double`, clase
arco
arco delimitado por un rectángulo
arco extendiéndose hasta un ángulo
ascendente
`BufferedImage`, clase
`closePath`, método de `GeneralPath`
color de fondo
`Color`, clase
`Component`, clase
componente vertical
contexto de gráficos
coordenada
coordenada *x*
coordenada *y*
descendente
`draw`, método
`draw3DRect`, método de `Graphics`
`drawArc`, método de `Graphics`
`drawLine`, método de `Graphics`
`drawOval`, método de `Graphics`
`drawPolygon`, método de `Graphics`
`drawPolyline`, método de `Graphics`
`drawRect`, método de `Graphics`
`drawRoundRect`, método de `Graphics`
eje *x*
eje *y*
`Ellipse2D.Double`, clase
estilo de tipo de letra
`fill`, método de `Graphics2D`
`fill3DRect`, método de `Graphics`
`fillArc`, método de `Graphics`
`fillOval`, método de `Graphics`
`fillPolygon`, método de `Graphics`
`fillRect`, método de `Graphics`
`fillRoundRect`, método de `Graphics`
`Font`, clase
`FontMetrics`, clase
`GeneralPath`, clase
`getAscent`, método de `FontMetrics`
`getBlue`, método de `Color`
`getDescent`, método de `FontMetrics`
`getFamily`, método de `Font`
`getFont`, método de `Graphics`
`getFontMetrics`, método de `Graphics`
`getGreen`, método de `Color`
`getHeight`, método de `FontMetrics`
`getLeading`, método de `FontMetrics`
`getName`, método de `Font`
`getRed`, método de `Color`
`getSize`, método de `Font`
`getStyle`, método de `Font`
`GradientPath`, clase
grado
grados negativos
grados positivos
`Graphics`, clase
`Graphics2D`, clase
interlineado
`isBold`, método de `Font`
`isItalic`, método de `Font`
`isPlain`, método de `Font`
Java2D, API
`Line2D.Double`, clase
línea base
`lineTo`, método de `GeneralPath`
métrica de los tipos de letra
`Monospaced`, tipo de letra
`moveTo`, método de `GeneralPath`
nombre de tipo de letra
objeto de gráficos
óvalo
`Paint`, interfaz
píxel
polígono
polígono cerrado
polígono relleno
`Polygon`, clase
punto
`Rectangle2D.Double`, clase
rectángulo delimitador
`rotate`, método de `Graphics2D`
`RoundRectangle2D.Double`, clase
`SansSerif`, tipo de letra
`Serif`, tipo de letra
`setColor`, método de `Graphics`
`setFont`, método de `Graphics`
`setPaint`, método de `Graphics2D`
`setStroke`, método de `Graphics2D`
`Shape`, interfaz
sistema de coordenadas
`Stroke`, interfaz
`SystemColor`, clase
`TexturePaint`, clase
tipo de letra
`translate`, método de `Graphics2D`
valor RGB

EJERCICIOS DE AUTOEVALUACIÓN

12.1 Complete las siguientes oraciones:

a) En Java2D, el método _____ de la clase _____ establece las características de una línea utilizada para dibujar una figura.
b) La clase _____ ayuda a especificar el relleno para una figura, de tal forma que el relleno cambie gradualmente de un color a otro.
c) El método _____ de la clase `Graphics` dibuja una línea entre dos puntos.
d) RGB son las iniciales de _____, _____ y _____.
e) Los tamaños de los tipos de letra se miden en unidades llamadas _____.
f) La clase _____ ayuda a especificar el relleno para una figura, utilizando un patrón dibujado en un objeto `BufferedImage`.

http://librosysolucionarios.net

12.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Los primeros dos argumentos del método `drawOval` de `Graphics` especifican la coordenada central del óvalo.
b) En el sistema de coordenadas de Java, los valores de *x* se incrementan de izquierda a derecha.
c) El método `fillPolygon` dibuja un polígono sólido en el color actual.
d) El método `drawArc` permite ángulos negativos.
e) El método `getSize` devuelve el tamaño del tipo de letra actual, en centímetros.
f) La coordenada de píxel *(0, 0)* se encuentra exactamente en el centro del monitor.

12.3 Encuentre el(los) error(es) en cada una de las siguientes instrucciones, y explique cómo corregirlos. Suponga que `g` es un objeto `Graphics`.
a) `g.setFont( "SansSerif" );`
b) `g.erase( x, y, w, h );  // borrar rectángulo en (x, y)`
c) `Font f = new Font( "Serif", Font.BOLDITALIC, 12 );`
d) `g.setColor( Color.Yellow );  // cambiar color a amarillo`

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

12.1 a) `setStroke`, `Graphics2D`. b) `GradientPath`. c) `drawLine`. d) Rojo, verde, azul. e) Puntos. f) `TexturePaint`.

12.2 a) Falso. Los primeros dos argumentos especifican la esquina superior izquierda del rectángulo delimitador.
b) Verdadero.
c) Verdadero.
d) Verdadero.
e) Falso. Los tamaños de los tipos de letra se miden en puntos.
f) Falso. La coordenada *(0,0)* corresponde a la esquina superior izquierda de un componente de la GUI, en el cual ocurre el dibujo.

12.3 a) El método `setFont` toma un objeto `Font` como argumento, no un `String`.
b) La clase `Graphics` no tiene un método `erase`. Debe utilizarse el método `clearRect`.
c) `Font.BOLDITALIC` no es un estilo de tipo de letra válido. Para obtener un tipo de letra en cursiva y negrita, use `Font.BOLD + Font.ITALIC`.
d) `Yellow` debe estar en mayúsculas, como en: `g.setColor( Color.YELLOW );`.

EJERCICIOS

12.4 Complete las siguientes oraciones:
a) La clase _____ de la API Java2D se utiliza para dibujar óvalos.
b) Los métodos `draw` y `fill` de la clase `Graphics2D` requieren un objeto de tipo _____ como su argumento.
c) Las tres constantes que especifican el estilo de los tipos de letra son _____, _____ y _____.
d) El método _____ de `Graphics2D` establece el color para pintar en figuras de Java2D.

12.5 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) El método `drawPolygon` conecta automáticamente los puntos de los extremos del polígono.
b) El método `drawLine` dibuja una línea entre dos puntos.
c) El método `fillArc` utiliza grados para especificar el ángulo.
d) En el sistema de coordenadas de Java, los valores de *y* se incrementan de arriba hacia abajo.
e) La clase `Graphics` hereda directamente de la clase `Object`.
f) La clase `Graphics` es una clase `abstract`.
g) La clase `Font` hereda directamente de la clase `Graphics`.

12.6 Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deberán estar separados por 10 píxeles. Use el método `drawOval` de la clase `Graphics`.

12.7 Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deberán estar separados por 10 píxeles. Use el método `drawArc`.

12.8 Modifique su solución al ejercicio 12.6, para dibujar los óvalos mediante el uso de instancias de la clase `Ellipse2D.Double` y el método `draw` de la clase `Graphics2D`.

12.9 Escriba un programa que dibuje líneas de longitudes aleatorias, en colores aleatorios.

12.10 Modifique su solución al ejercicio 12.9 para dibujar líneas aleatorias en colores aleatorios y grosores de línea aleatorios. Use la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D` para dibujar las líneas.

12.11 Escriba un programa que muestre triángulos generados al azar en distintos colores. Cada triángulo deberá rellenarse con un color distinto. Use la clase `GeneralPath` y el método `fill` de la clase `Graphics2D` para dibujar los triángulos.

12.12 Escriba un programa que dibuje caracteres al azar, en distintos tamaños y colores de tipo de letra.

12.13 Escriba un programa que dibuje una cuadrícula de 8 por 8. Use el método `drawLine`.

12.14 Modifique su solución al ejercicio 12.13 para dibujar la cuadrícula utilizando instancias de la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D`.

12.15 Escriba un programa que dibuje una cuadrícula de 10 por 10. Use el método `drawRect`.

12.16 Modifique su solución al ejercicio 12.15 para dibujar la cuadrícula utilizando instancias de la clase `Rectangle2D.Double` y el método `draw` de la clase `Graphics2D`.

12.17 Escriba un programa que dibuje un tetraedro (una figura tridimensional con cuatro caras triangulares). Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

12.18 Escriba un programa que dibuje un cubo. Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

12.19 En el ejercicio 3.10 usted escribió un applet que recibía como entrada, por parte del usuario, el radio de un círculo y que mostraba el diámetro, circunferencia y área de ese círculo. Modifique su solución al ejercicio 3.10 para leer un conjunto de coordenadas además del radio. Después dibuje el círculo y muestre su diámetro, circunferencia y área utilizando un objeto `Ellipse2D.Double` para representar el círculo y el método `draw` de la clase `Graphics2D` para mostrar el círculo.

12.20 Escriba una aplicación que simule un protector de pantalla. La aplicación deberá dibujar líneas al azar, utilizando el método `drawLine` de la clase `Graphics`. Después de dibujar 100 líneas, la aplicación deberá borrarse a sí misma y empezar a dibujar líneas nuevamente. Para permitir al programa dibujar en forma continua, coloque una llamada a `repaint` como la última línea en el método `paint`. ¿Observó algún problema con esto en su sistema?

12.21 He aquí un adelanto. El paquete `javax.swing` contiene una clase llamada `Timer`, la cual es capaz de llamar al método `actionPerformed` de la interfaz `ActionListener` durante un intervalo de tiempo fijo (especificado en milisegundos). Modifique su solución al ejercicio 12.20 para eliminar la llamada a `repaint` desde el método `paint`. Declare su clase de manera que implemente a `ActionListener`. (El método `actionPerformed` deberá simplemente llamar a `repaint`). Declare una variable de instancia de tipo `Timer`, llamada `temporizador`, en su clase. En el constructor para su clase, escriba las siguientes instrucciones:

```
temporizador = new Timer( 1000, this );
temporizador.start();
```

Esto crea una instancia de la clase `Timer` que llamará al método `actionPerformed` del objeto `this` cada 1000 milisegundos (es decir, cada segundo).

12.22 Modifique su solución al ejercicio 12.21 para permitir al usuario introducir el número de líneas aleatorias que deben dibujarse antes de que la aplicación se borre a sí misma y empiece a dibujar líneas otra vez. Use un objeto `JTextField` para obtener el valor. El usuario deberá ser capaz de escribir un nuevo número en el objeto `JTextField` en cualquier momento durante la ejecución del programa. Use una clase interna para realizar el manejo de eventos para el objeto `JTextField`.

12.23 Modifique su solución al ejercicio 12.21, de tal forma que utilice la generación de números aleatorios para seleccionar diferentes figuras a mostrar. Use métodos de la clase `Graphics`.

12.24 Modifique su solución al ejercicio 12.23 para utilizar clases y herramientas de dibujo de la API Java2D. Para las figuras como rectángulos y elipses, dibújelas con degradados generados al azar. Use la clase `GradientPaint` para generar el degradado.

12.25 Modifique su solución al ejercicio 7.21 (*Gráficos de tortuga*) para agregar una interfaz gráfica de usuario, mediante el uso de objetos `JTextField` y `JButton`. Además, dibuje líneas en vez de asteriscos (*). Cuando el programa de gráficos de tortuga especifique un movimiento, traduzca el número de posiciones en un número de píxeles en la pantalla, multiplicando el número de posiciones por 10 (o cualquier valor que usted elija). Implemente el dibujo con características de la API Java2D.

12.26 Implemente una versión gráfica del problema del Paseo del caballo (ejercicios 7.22, 7.23 y 7.26). A medida que se realice cada movimiento, la celda apropiada del tablero de ajedrez deberá actualizarse con el número de movimien-

http://librosysolucionarios.net

to apropiado. Si el resultado del programa es un *paseo completo* o un *paseo cerrado*, el programa deberá mostrar un mensaje apropiado. Si lo desea, puede utilizar la clase `Timer` (vea el ejercicio 11.24) para que le ayude con la animación del Paseo del caballo. Cada segundo deberá realizarse el siguiente movimiento.

12.27 Produzca una versión gráfica de la simulación *La tortuga y la liebre* (ejercicio 7.15). Simule la montaña dibujando un arco que se extienda desde la parte inferior izquierda de la ventana, hasta la parte superior derecha. La tortuga y la liebre deberán correr hacia arriba de la montaña. Implemente la salida gráfica de manera que la tortuga y la liebre se impriman en el arco, en cada movimiento. [*Nota:* Extienda la longitud de la carrera de 70 a 300, para que cuente con un área de gráficos más grande.]

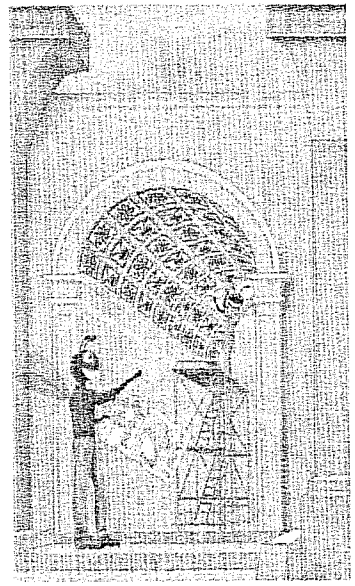
12.28 Escriba un programa que utilice el método `drawPolyline` para dibujar una espiral.

12.29 Escriba un programa que reciba como entrada cuatro números y que los grafique en forma de gráfico de pastel. Use la clase `Arc2D.Double` y el método `fill` de la clase `Graphics2D` para realizar el dibujo. Dibuje cada pieza del pastel en un color distinto.

12.30 Escriba un applet que reciba como entrada cuatro números y que los grafique en forma de gráfico de barras. Utilice la clase `Rectangle2D.Double` y el método `fill` de la clase `Graphics2D` para realizar el dibujo. Dibuje cada barra en un color distinto.

13

Componentes de la interfaz gráfica de usuario: Parte 1



Objetivos

- Comprender los principios de diseño de las interfaces gráficas de usuario (GUI).
- Crear interfaces gráficas de usuario.
- Aprender acerca de los paquetes que contienen componentes relacionados con las GUIs, clases e interfaces para manejo de eventos.
- Crear y manipular botones, etiquetas, listas, campos de texto y paneles.
- Entender los eventos de ratón y los eventos de teclado.
- Aprender a utilizar los administradores de esquemas.

... los profetas más sabios se aseguran primero del evento.
Horace Walpole

¿Crees que puedo escuchar todo el día esas cosas?
Lewis Carroll

Habla en afirmativo; enfatiza tu elección ignorando todo lo que rechaces.
Ralph Waldo Emerson

Tú pagas, por lo tanto, tú decides.
Punch

Adivina si puedes, elige si te atreves.
Pierre Corneille

¡Todo aquel que entre aquí, abandone toda esperanza!
Dante Alighieri

Salida, perseguida por un oso.
William Shakespeare

http://librosysolucionarios.net

Plan general

13.1 Introducción
13.2 Generalidades de los componentes de Swing
13.3 JLabel
13.4 Manejo de eventos
13.5 Campos de texto
13.6 Cómo funciona el manejo de eventos
13.7 JButton
13.8 JCheckBox y JRadioButton
13.9 JComboBox
13.10 JList
13.11 Listas de selección múltiple
13.12 Manejo de eventos de ratón
13.13 Clases adaptadoras
13.14 Manejo de eventos de teclas
13.15 Administradores de esquemas
13.15.1 FlowLayout
13.15.2 BorderLayout
13.15.3 GridLayout
13.16 Paneles
13.17 (Ejemplo práctico opcional) Acerca de los objetos: Casos de uso

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

13.1 Introducción

Una *interfaz gráfica de usuario (GUI)* presenta un mecanismo amigable al usuario para interactuar con un programa; ya que a este último le proporcina una "apariencia visual" única. Al proporcionar distintos programas en los que los componentes de la interfaz de usuario sean consistentes e intuitivos, los usuarios pueden familiarizarse con un programa incluso antes de utilizarlo. Esto, a su vez, reduce el tiempo que requieren para aprender a usar un programa y aumenta su habilidad para utilizar el programa en una manera productiva.



Observación de apariencia visual 13.1

Las interfaces de usuario consistentes permiten a un usuario aprender, con más rapidez, a utilizar nuevas aplicaciones.

Como ejemplo de una GUI, en la figura 13.1 se muestra una ventana del navegador Web Microsoft Internet Explorer, con algunos componentes de la GUI etiquetados. En la ventana hay una *barra de menús* que contiene *menús* (**Archivo**, **Edición**, **Ver**, etcétera). Debajo de la barra de menús hay un conjunto de *botones*, cada uno tiene una tarea definida en Internet Explorer. A la derecha de los botones hay un *cuadro combinado*, en el cual el usuario puede escribir el nombre de un sitio Web a visitar, o puede hacer clic en la flecha hacia abajo del lado derecho del cuadro combinado para ver una lista de los sitios visitados previamente. Los menús, botones y el cuadro combinado son parte de la GUI de Internet Explorer. Éstos le permiten interactuar con este navegador Web. En este capítulo y en el siguiente, demostraremos el uso de muchos de los componentes de la GUI que permiten a los usuarios interactuar con sus programas.

Las GUIs se crean a partir de *componentes de la GUI* (también conocidos como *controles* o *widgets* [accesorios de ventana]). Un componente de la GUI es un objeto con el cual interactúa el usuario mediante el ratón, el teclado u otra forma de entrada, como el reconocimiento de voz. Varios de los componentes de la GUI de Java se muestran en la figura 13.2. En las siguientes secciones hablaremos detalladamente sobre cada uno de estos componentes.

http://librosysolucionarios.net

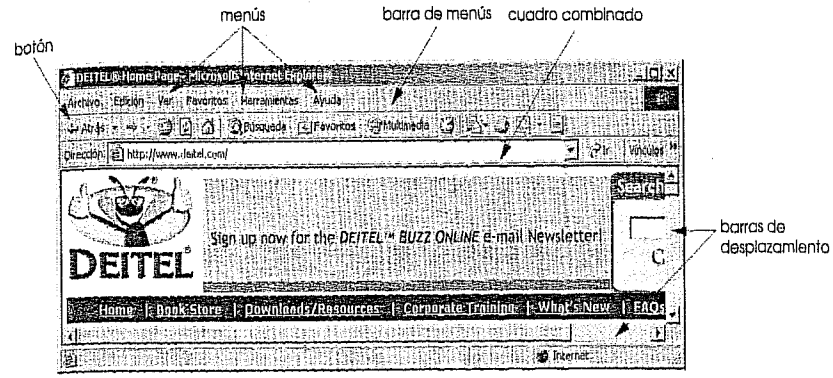


Figura 13.1 Ventana de Microsoft Internet Explorer con sus componentes de la GUI.

13.2 Generalidades de los componentes de Swing

Las clases que crean los componentes de la GUI que se muestran en la figura 13.2 son algunos de los *componentes de la GUI de Swing* del paquete `javax.swing`. Estos componentes de la GUI se hicieron estándar en Java cuando se liberó la Plataforma Java 2 versión 1.2. La mayoría de los *componentes de Swing*, como se les llama comúnmente, están escritos, se manipulan y muestran completamente en Java (los denominados componentes *puros de Java*). Los componentes de Swing son parte de la *JFC (Java Foundation Classes)*, las bibliotecas de Java para el desarrollo de GUIs para múltiples plataformas. Para obtener información completa sobre la JFC, visite la página `java.sun.com/products/jfc`.

Los componentes originales de la GUI del paquete `java.awt` del *AWT (Abstract Windows Toolkit)* están enlazados directamente con las herramientas de interfaz gráfica de usuario de la plataforma local. Cuando se ejecuta un programa de Java con una GUI del AWT en distintas plataformas Java, los componentes de la GUI del programa se muestran de manera distinta en cada plataforma. Considere como ejemplo un programa que muestra un objeto de tipo `Button` (paquete `java.awt`). En una computadora que ejecuta el sistema operativo Microsoft Windows, el objeto `Button` tendrá la misma apariencia visual que los botones de otras aplicaciones Windows. De manera similar, en una computadora que ejecuta el sistema operativo Mac OS, de Apple, el objeto `Button` tendrá la misma apariencia visual que los botones de otras aplicaciones Macintosh. En ocasiones, la forma en que un usuario puede interactuar con un componente AWT específico es distinta entre las plataformas.

Componente	Descripción
`JLabel`	Área en donde pueden mostrarse íconos o texto no editable.
`JTextField`	Área en la que el usuario introduce datos desde el teclado. Esta área también puede mostrar información.
`JButton`	Área que, cuando el ratón hace clic sobre ella, desencadena un evento.
`JCheckBox`	Componente de la GUI que puede estar o no seleccionado.
`JComboBox`	Lista desplegable de elementos, de los cuales el usuario puede seleccionar uno haciendo clic en él o posiblemente escribiendo dentro del cuadro.
`JList`	Área que contiene una lista de elementos, de los cuales el usuario puede seleccionar cualquiera haciendo clic en él. Pueden seleccionarse varios elementos.
`JPanel`	Contenedor en el cual pueden colocarse y organizarse componentes.

Figura 13.2 Algunos componentes de la GUI básicos.

En conjunto, la apariencia y la manera en que el usuario interactúa con el programa se conocen como la *vista y el sentido* de ese programa. Los componentes de Swing permiten al programador especificar una vista y sentido uniformes entre todas las plataformas, o proporcionar una vista y sentido personalizados para cada plataforma. Un programa puede incluso cambiar la vista y el sentido durante su ejecución. Por ejemplo, un programa podría permitir a los usuarios elegir su vista y sentido preferidos.

Observación de apariencia visual 13.2

Los componentes de Swing se implementan en Java, por lo que proporcionan un mayor nivel de portabilidad y flexibilidad que los componentes originales de la GUI del paquete `java.awt`*, que utilizaban los componentes de la GUI de la plataforma subyacente.*

Los componentes de Swing comúnmente se conocen como componentes ligeros; están escritos completamente en Java, por lo que no son "abrumados" por las complejas herramientas de la GUI de la plataforma en que se utilicen. Sin embargo, a los componentes de AWT (muchos de los cuales se comparan con los componentes de Swing), que están enlazados a la plataforma local, se les llama componentes pesados; éstos dependen del sistema de ventanas de la plataforma local para determinar su funcionalidad y su vista y sentido. Cada componente pesado tiene un igual (del paquete `java.awt.peer`) que es responsable de las interacciones entre el componente y la plataforma local que muestra y manipula a ese componente. Varios componentes de Swing son componentes pesados, como las subclases de `java.awt.Window` (por ejemplo, `JFrame`) que muestran ventanas en la pantalla, y las subclases de `java.applet.Applet` (como `JApplet`). Estos componentes aún requieren de una interacción directa con el sistema de ventanas local. Como tal, la apariencia y funcionalidad de un componente pesado de la GUI de Swing están restringidas por el sistema de ventanas local. Como veremos, los componentes ligeros proporcionan un mayor control sobre su apariencia y funcionalidad.

Tip de portabilidad 13.1

La vista de una GUI definida con componentes pesados de la GUI del paquete `java.awt` *puede variar entre plataformas. Los componentes pesados se enlazan en la GUI de la plataforma local, la cual varía de plataforma en plataforma.*

En la figura 13.3 se muestra una jerarquía de herencia, la cual contiene clases que declaran atributos y comportamientos comunes para la mayoría de los componentes de Swing. La clase `Object` es la superclase de la jerarquía de clases de Java. La clase `Component` (paquete `java.awt`) es una subclase de `Object`; la clase `Container` (paquete `java.awt`) es una subclase de `Component`; y la clase `JComponent` (paquete `javax.swing`) es una subclase de `Container`. Una clase que extiende a la clase `Component` *es una* `Component`. Por ejemplo, la clase `Container` extiende a la clase `Component` y la clase `Component` extiende a `Object`. Por lo tanto, una `Container` *es una* `Component` y *es una* `Object`, y una `Component` *es una* `Object`. Una clase que extiende a la clase `Container` *es una* `Container`. Por lo tanto, una `JComponent` *es una* `Container`. Las clases `Component` pueden organizarse mediante clases `Container`. Como una clase `Container` *es una* `Component`, una GUI puede estructurarse con clases `Container` adjuntas a otras clases `Container`.

Observación de ingeniería de software 13.1

Para utilizar componentes de la GUI efectivamente, las jerarquías de herencia de `javax.swing` *y* `java.awt` *deben comprenderse; especialmente la clase* `Component`*, la clase* `Container` *y la clase* `JComponent`*, que declaran características comunes para la mayoría de los componentes de Swing.*

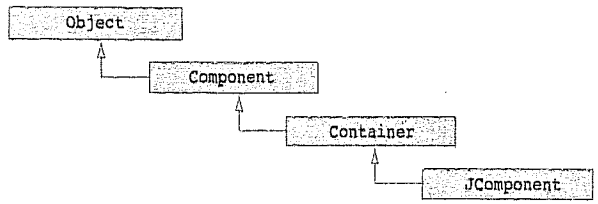


Figura 13.3 Superclases comunes de muchos de los componentes de Swing.

La clase `Component` declara los atributos y comportamientos comunes de todas las subclases de `Component`. Con pocas excepciones, la mayoría de los componentes de la GUI extienden a la clase `Component` en forma directa o indirecta. Un método que se origina en la clase `Component` y que hemos utilizado frecuentemente en subprogramas (applets) es `paint`. Otro método descrito previamente que se origina en `Component` es `repaint`. Es importante comprender los métodos de la clase `Component`, ya que las operaciones comunes para la mayoría de los componentes de la GUI (tanto de Swing como de AWT) se encuentran en la clase `Component`.

Buena práctica de programación 13.1

Estudie los métodos de la clase `Component` *en la documentación en línea del SDK de Java 2 (*`java.sun.com/j2se/1.4.1/docs/api/java/awt/Component.html`*), para conocer las herramientas comunes para la mayoría de los componentes de la GUI.*

Una clase `Container` administra una colección de componentes relacionados. En aplicaciones con objetos `JFrame` y en subprogramas, adjuntamos componentes al panel de contenido, que es un objeto de la clase `Container`. Esta clase declara los atributos y comportamientos comunes para todas las subclases de `Container`. Un método que se origina en la clase `Container` es `add`, el cual se utiliza para adjuntar componentes al panel de contenido (un objeto `Container`). Otro método que se origina en la clase `Container` es `setLayout`, el cual permite a un programa especificar el administrador de esquemas que ayuda a un `Container` a posicionar y ajustar el tamaño de sus componentes.

Buena práctica de programación 13.2

Estudie los métodos de la clase `Container` *en la documentación en línea del SDK de Java 2 (*`java.sun.com/j2se/1.4.1/docs/api/java/awt/Container.html`*), para conocer las herramientas comunes para todos los contenedores de los componentes de la GUI.*

La clase `JComponent` es la superclase de la mayoría de los componentes de Swing. Esta clase declara los atributos y comportamientos comunes de todas las subclases de `JComponent`, incluyendo:

1. una *apariencia visual adaptable*, la cual puede utilizarse para personalizar la apariencia de los componentes (por ejemplo, para usarlos en plataformas específicas);
2. teclas de método abreviado (llamadas *nemónicos*) para un acceso directo a los componentes de la GUI por medio del teclado;
3. herramientas para manejo de eventos comunes, para casos en los que varios componentes de la GUI inician las mismas acciones en un programa;
4. breves descripciones del propósito de un componente de la GUI (lo que se conoce como *cuadros de información sobre herramientas o tool tips*) que se muestran cuando el cursor del ratón se coloca sobre el componente durante un breve periodo;
5. soporte para tecnologías de ayuda, como lectores de pantalla Braille para las personas con impedimentos visuales;
6. soporte para la *localización* de la interfaz de usuario; es decir, personalizar la interfaz de usuario para mostrarla en distintos lenguajes y utilizar las convenciones de la cultura local.

Éstas son sólo algunas de las muchas características de los componentes de Swing. En este capítulo y en el 14 hablaremos sobre varias de estas características.

Buena práctica de programación 13.3

Estudie los métodos de la clase `JComponent` *en la documentación en línea del SDK de Java 2 (*`java.sun.com/j2se/1.4.1/docs/api/javax/swing/JComponent.html`*), para conocer las herramientas comunes para los componentes de Swing.*

13.3 `JLabel`

Las *etiquetas* proporcionan instrucciones de texto o información en una GUI. Las etiquetas se definen mediante la clase `JLabel`, una subclase de `JComponent`. Una etiqueta muestra una sola línea de *texto de sólo lectura*, una imagen o texto con una imagen. Los programas raras veces modifican el contenido de una etiqueta después de crearla. La aplicación de la figura 13.4 demuestra varias características de `JLabel`.

http://librosysolucionarios.net

Buena práctica de programación 13.4

Estudie los métodos de la clase `JLabel` *en la documentación en línea del SDK de Java 2 (*`java.sun.com/j2se/1.4.1/docs/api/javax/swing/JLabel.html`*), para conocer las herramientas completas de la clase antes de usarla.*

El programa declara tres objetos `JLabel` en la línea 8. Los objetos `JLabel` se instancian en el constructor de `PruebaEtiqueta` (líneas 11 a 43). En la línea 20 se crea un objeto `JLabel` con el texto `"Etiqueta con texto"`. La etiqueta muestra este texto cuando aparece en la pantalla (es decir, cuando se muestra la ventana `PruebaEtiqueta` en este programa).

```
1  // Fig. 13.4: PruebaEtiqueta.java
2  // Demostración de la clase JLabel.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaEtiqueta extends JFrame {
8     private JLabel etiqueta1, etiqueta2, etiqueta3;
9
10    // configurar la GUI
11    public PruebaEtiqueta()
12    {
13       super( "Prueba de JLabel" );
14
15       // obtener panel de contenido y establecer su esquema
16       Container contenedor = getContentPane();
17       contenedor.setLayout( new FlowLayout() );
18
19       // constructor de JLabel con un argumento cadena
20       etiqueta1 = new JLabel( "Etiqueta con texto" );
21       etiqueta1.setToolTipText( "Esta es la etiqueta1" );
22       contenedor.add( etiqueta1 );
23
24       // constructor de JLabel con argumentos cadena, Icono y alineación
25       Icon insecto = new ImageIcon( "insecto1.gif" );
26       etiqueta2 = new JLabel( "Etiqueta con texto e icono", insecto,
27          SwingConstants.LEFT );
28       etiqueta2.setToolTipText( "Esta es la etiqueta2" );
29       contenedor.add( etiqueta2 );
30
31       // constructor de JLabel sin argumentos
32       etiqueta3 = new JLabel();
33       etiqueta3.setText( "Etiqueta con icono y texto en parte inferior" );
34       etiqueta3.setIcon( insecto );
35       etiqueta3.setHorizontalTextPosition( SwingConstants.CENTER );
36       etiqueta3.setVerticalTextPosition( SwingConstants.BOTTOM );
37       etiqueta3.setToolTipText( "Ésta es la etiqueta3" );
38       contenedor.add( etiqueta3 );
39
40       setSize( 275, 170 );
41       setVisible( true );
42
43    } // fin del constructor
44
45    public static void main( String args[] )
```

Figura 13.4 Objetos `JLabel` con texto e iconos. (Parte 1 de 2.)

http://librosysolucionarios.net


```
46    {
47       PruebaEtiqueta aplicacion = new PruebaEtiqueta();
48       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
49    }
50
51 } // fin de la clase PruebaEtiqueta
```

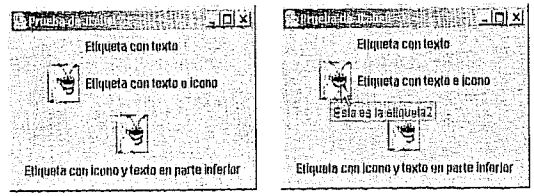


Figura 13.4 Objetos `JLabel` con texto e iconos. (Parte 2 de 2.)

En la línea 21 se utiliza el método `setToolTipText` (heredado por `JLabel` de `JComponent`) para especificar la información sobre herramientas (vea la segunda captura de pantalla de la figura 13.4) que se muestra cuando el usuario coloca el cursor del ratón sobre la etiqueta en la GUI. Cuando ejecute este programa, pruebe a colocar el ratón sobre cada etiqueta para ver su información sobre herramientas. En la línea 22 se agrega la `etiqueta1` al panel de contenido de la ventana `PruebaEtiqueta`.

Observación de apariencia visual 13.3

Use cuadros de información sobre herramientas para agregar texto descriptivo a los componentes de la GUI. Este texto ayuda al usuario a determinar el propósito del componente de la GUI en la interfaz de usuario.

Varios componentes de Swing pueden mostrar imágenes tanto con un argumento `Icon` para su constructor o por medio de un método que se conoce normalmente como `setIcon`. Un `Icon` es un objeto de cualquier clase que implemente a la interfaz `Icon` (paquete `javax.swing`). Una de esas clases es `ImageIcon` (paquete `javax.swing`), la cual soporta varios formatos de imagen, incluyendo *GIF (Formato de intercambio de gráficos)*, *PNG (Gráficos portables de red)* y *JPEG (Grupo unido de expertos en fotografía)*. Los nombres de archivo para cada uno de estos tipos terminan con `.gif`, `.png` o `.jpg` (o `.jpeg`), respectivamente. Hablaremos sobre las imágenes con más detalle en el capítulo 19. En la línea 25 se declara un objeto `ImageIcon`. El archivo `insecto1.gif` contiene la imagen a cargar y almacenar en el objeto `ImageIcon`. Por ahora supondremos que el archivo se encuentra en el mismo directorio que el programa. (Hablaremos sobre cómo colocar el archivo de imagen en cualquier otro lado en el capítulo 19.) El objeto `ImageIcon` se asigna a la referencia `Icon` llamada `insecto`. Recuerde, la clase `ImageIcon` implementa a la interfaz `Icon`; por lo tanto, un `ImageIcon` *es un* `Icon`.

Un objeto `JLabel` puede mostrar un `Icon`. En las líneas 26 y 27 se utiliza otro constructor de `JLabel` para crear una etiqueta que muestra el texto `"Etiqueta con texto e icono"` y el `Icono` al que `insecto` hace referencia. El último argumento para el constructor indica que el contenido de la etiqueta se debe *justificar a la izquierda*, o *alinear a la izquierda* (es decir, el icono y el texto se encuentran del lado izquierdo del área de la etiqueta en la pantalla). La interfaz `SwingConstants` (paquete `javax.swing`) declara un conjunto de constantes enteras comunes (como `SwingConstants.LEFT`) que se utilizan con muchos componentes de Swing. De manera predeterminada, el texto aparece a la derecha de la imagen cuando una etiqueta contiene tanto texto como una imagen. Las alineaciones horizontal y vertical de una etiqueta pueden establecerse con los métodos `setHorizontalAlignment` y `setVerticalAlignment`, respectivamente. En la línea 28 se especifica el texto de información sobre herramientas para la `etiqueta2`, y en la línea 29 se agrega `etiqueta2` al panel de contenido.

Error común de programación 13.1

Si usted no agrega explícitamente un componente de la GUI a un contenedor, el componente de la GUI no se mostrará cuando el contenedor aparezca en la pantalla.



Error común de programación 13.2

Una excepción `NullPointerException` *ocurre cuando un programa invoca al método* `add` *de* `Container` *con una referencia* `null` *como argumento.*

La clase `JLabel` proporciona muchos métodos para cambiar la apariencia de una etiqueta, una vez que se ha creado su instancia. En la línea 32 se crea un objeto `JLabel` y se invoca su constructor sin argumentos. Dicha etiqueta al principio no tiene texto ni `Icon`. En la línea 33 se utiliza el método `setText` de `JLabel` para establecer el texto que se muestra en la etiqueta. El método `getText` correspondiente recupera el texto que se muestra actualmente en una etiqueta. En la línea 34 se utiliza el método `setIcon` de `JLabel` para especificar el objeto `Icon` a mostrar en la etiqueta. El método `getIcon` correspondiente recupera el objeto `Icon` mostrado actualmente en una etiqueta. En las líneas 35 y 36 se utilizan los métodos `setHorizontalTextPosition` y `setVerticalTextPosition` de `JLabel` para especificar la posición del texto en la etiqueta. En este caso, el texto se centrará horizontalmente y aparecerá en la parte inferior de la etiqueta. Por lo tanto, el objeto `Icon` aparecerá encima del texto. Las constantes de posición horizontal en `SwingConstants` son `LEFT`, `CENTER` y `RIGHT`. Las constantes de posición vertical en `SwingConstants` son `TOP`, `CENTER` y `BOTTOM`. En la línea 37 se establece el texto de información sobre herramientas para la `etiqueta3`. En la línea 38 se agrega `etiqueta3` al panel de contenido.

13.4 Manejo de eventos

En la sección anterior no hablamos sobre el manejo de eventos, ya que los usuarios no interactúan con objetos `JLabel`. Las GUIs están *controladas por eventos* (es decir, generan *eventos* cuando el usuario del programa interactúa con la GUI). Algunos eventos comunes (interacciones) son mover el ratón, hacer clic con los botones del ratón, hacer clic en un botón de la pantalla, escribir en un campo de texto, seleccionar un elemento de un menú y cerrar una ventana. Cuando ocurre una interacción con el usuario, se envía un mensaje al programa. La información de los eventos de la GUI se almacena en un objeto de una clase que extiende a `AWTEvent`. En la figura 13.5 se muestra una jerarquía que contiene muchas de las clases de eventos del paquete `java.awt.event`. Algunas de estas clases de eventos se describirán en este capítulo y en el 14. Los tipos de eventos del paquete `java.awt.event` se utilizan con componentes de AWT y de Swing. Los tipos de eventos adicionales, específicos para Swing, se declaran en el paquete `javax.swing.event`.

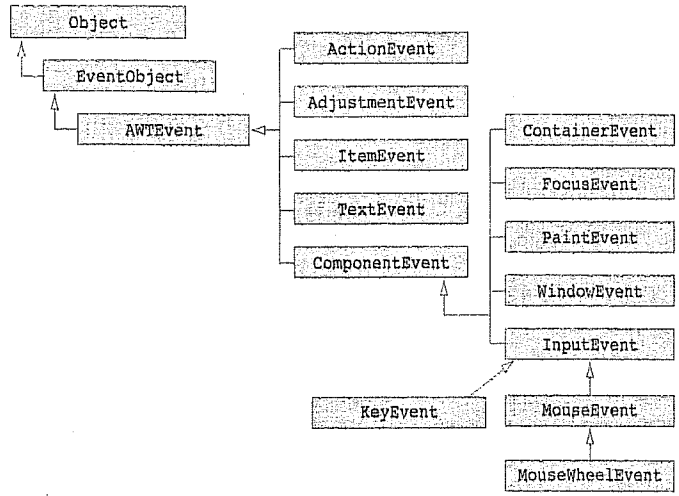


Figura 13.5 Algunas clases de eventos del paquete `java.awt.event`.

El mecanismo de manejo de eventos consta de tres partes: el *origen del evento*, el *objeto del evento* y el *componente de escucha del evento*. El origen del evento es el componente de la GUI específico con el cual interactúa el usuario. El objeto del evento encapsula la información acerca del evento que ocurrió. Esta información incluye una referencia al origen del evento y cualquier información específica de ese evento que pueda ser requerida por el componente de escucha del evento, para poder manejarlo. El componente de escucha del evento es un objeto que recibe la notificación del origen del evento cuando éste ocurre; en efecto, "escucha" a la espera de un evento y se ejecuta en respuesta a ese evento. El componente de escucha del evento recibe un objeto del evento cuando se le notifica que ocurrió éste. Después utiliza el objeto para responder al evento. El origen del evento se requiere para proveer métodos que permitan a los componentes de escucha ser registrados o no registrados. El origen del evento mantiene una lista de sus componentes de escucha registrados, y notifica a cada uno de ellos cuando ocurre un evento.

El programador debe realizar dos tareas clave para procesar un evento de interfaz gráfica de usuario en un programa: debe registrar un *componente de escucha del evento* para el componente de la GUI que se espera genere ese evento, y debe implementar un *método manejador del evento* (o un conjunto de métodos manejadores del evento). Los métodos manejadores de eventos se conocen comúnmente como *manejadores de eventos*. Un componente de escucha para un evento de la GUI es un objeto de una clase que implementa a una o más de las interfaces de componentes de escucha de eventos de los paquetes `java.awt.event` y `javax.swing.event`. Muchos de los tipos de componentes de escucha de eventos son comunes para los componentes Swing y AWT. Dichos tipos se declaran en el paquete `java.awt.event` y muchos de ellos se muestran en la figura 13.6. Los tipos de componentes de escucha de eventos adicionales que son específicos para los componentes Swing se declaran en el paquete `javax.swing.event`.

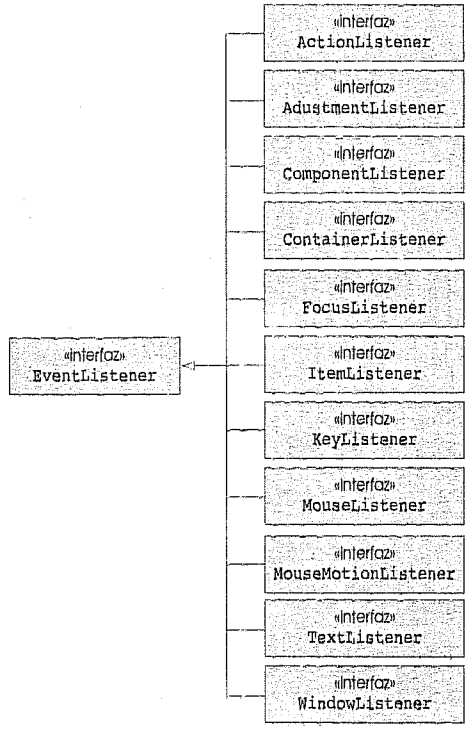


Figura 13.6 Interfaces de componentes de escucha de eventos del paquete `java.awt.event`.

http://librosysolucionarios.net

Cada interfaz de componente de escucha de eventos especifica uno o más métodos manejadores de eventos que *deben* declararse en la clase que implemente a la interfaz de componente de escucha de eventos. Recuerde que en la sección 10.8 vimos que cualquier clase que implementa a una interfaz debe declarar todos los métodos **abstract** de esa interfaz; de lo contrario, la clase se convierte en clase **abstract** y no puede utilizarse para crear objetos. El uso de componentes de escucha de eventos en el manejo de eventos se conoce como el *modelo de delegación de eventos*: el procesamiento de un evento se delega a un objeto específico (el componente de escucha) en el programa.

Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus componentes de escucha registrados, llamando al método manejador de eventos apropiado de cada componente de escucha. Por ejemplo, cuando el usuario oprime *Intro* en un objeto **JTextField**, se hace una llamada al método **actionPerformed** del componente de escucha registrado. ¿Cómo se registró el manejador de eventos? ¿Cómo sabe el componente de la GUI que debe llamar a **actionPerformed** en vez de llamar a otro método manejador de eventos? Responderemos estas preguntas y haremos un diagrama de la interacción como parte del siguiente ejemplo.

13.5 Campos de texto

Los objetos ***JTextField*** y ***JPasswordField*** (paquete **javax.swing**) son áreas de una sola línea en las que el usuario puede introducir texto mediante el teclado, o el programa puede mostrar texto. Un objeto **JPasswordField** muestra que los caracteres se están escribiendo a medida que el usuario los introduce, pero oculta los caracteres actuales, suponiendo que representan una contraseña que no debe conocer nadie más aparte del usuario. Cuando éste escribe datos en uno de estos campos y oprime *Intro*, ocurre un evento de acción. Si el programa registra un componente de escucha de eventos, este componente procesa el evento y el programa puede utilizar los datos en el campo al momento del evento. La clase **JTextField** extiende a la clase ***JTextComponent*** (paquete **javax.swing.text**), la cual proporciona muchas características comunes a los componentes basados en texto de Swing. La clase **JPasswordField** extiende a **JTextField** y agrega varios métodos específicos para el procesamiento de contraseñas.

La aplicación de la figura 13.7 utiliza las clases **JTextField** y **JPasswordField** para crear y manipular cuatro campos de texto. Cuando el usuario oprime *Intro* en el campo activo, se muestra un cuadro de diálogo de mensaje que contiene el texto en el campo. El campo activo se dice que está "*enfocado*". Uno puede cambiar el enfoque a cualquier componente, haciendo clic en ese componente o llamando al método **request-**

```
1  // Fig. 13.7: PruebaCampoTexto.java
2  // Demostración de la clase JTextField.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaCampoTexto extends JFrame {
8     private JTextField campoTexto1, campoTexto2, campoTexto3;
9     private JPasswordField campoContrasenia;
10
11    // configurar la GUI
12    public PruebaCampoTexto()
13    {
14       super( "Prueba de JTextField y JPasswordField" );
15
16       Container contenedor = getContentPane();
17       contenedor.setLayout( new FlowLayout() );
18
19       // crear campo de texto con tamaño predeterminado
20       campoTexto1 = new JTextField( 10 );
21       contenedor.add( campoTexto1 );
```

Figura 13.7 Objetos **JTextField** y **JPasswordField**. (Parte 1 de 3.)


```
22
23       // crear campo de texto con texto predeterminado
24       campoTexto2 = new JTextField( "Escriba el texto aquí" );
25       contenedor.add( campoTexto2 );
26
27       // crear campo de texto con texto predeterminado,
28       // 20 elementos visibles y sin manejador de eventos
29       campoTexto3 = new JTextField( "Campo de texto no editable", 20 );
30       campoTexto3.setEditable( false );
31       contenedor.add( campoTexto3 );
32
33       // crear campo de contraseña con texto predeterminado
34       campoContrasenia = new JPasswordField( "Texto oculto" );
35       contenedor.add( campoContrasenia );
36
37       // registrar manejadores de eventos
38       ManejadorCampoTexto manejador = new ManejadorCampoTexto();
39       campoTexto1.addActionListener( manejador );
40       campoTexto2.addActionListener( manejador );
41       campoTexto3.addActionListener( manejador );
42       campoContrasenia.addActionListener( manejador );
43
44       setSize( 325, 100 );
45       setVisible( true );
46
47    } // fin del constructor de PruebaCampoTexto
48
49    public static void main( String args[] )
50    {
51       PruebaCampoTexto aplicacion = new PruebaCampoTexto();
52       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
53    }
54
55    // clase interna privada para el manejo de eventos
56    private class ManejadorCampoTexto implements ActionListener {
57
58       // procesar eventos de campo de texto
59       public void actionPerformed( ActionEvent evento )
60       {
61          String cadena = "";
62
63          // el usuario oprimió Intro en objeto JTextField campoTexto1
64          if ( evento.getSource() == campoTexto1 )
65             cadena = "campoTexto1: " + evento.getActionCommand();
66
67          // el usuario oprimió Intro en objeto JTextField campoTexto2
68          else if ( evento.getSource() == campoTexto2 )
69             cadena = "campoTexto2: " + evento.getActionCommand();
70
71          // el usuario oprimió Intro en objeto JTextField campoTexto3
72          else if ( evento.getSource() == campoTexto3 )
73             cadena = "campoTexto3: " + evento.getActionCommand();
74
75          // el usuario oprimió Intro en objeto JTextField campoContrasenia
76          else if ( evento.getSource() == campoContrasenia ) {
```

Figura 13.7 Objetos **JTextField** y **JPasswordField**. (Parte 2 de 3.)

http://librosysolucionarios.net

```
77                cadena = "campoContrasenia: " +
78                   new String( campoContrasenia.getPassword() );
79             }
80
81             JOptionPane.showMessageDialog( null, cadena );
82
83          } // fin del método actionPerformed
84
85       } // fin de la clase interna privada ManejadorCampoTexto
86
87    } // fin de la clase PruebaCampoTexto
```

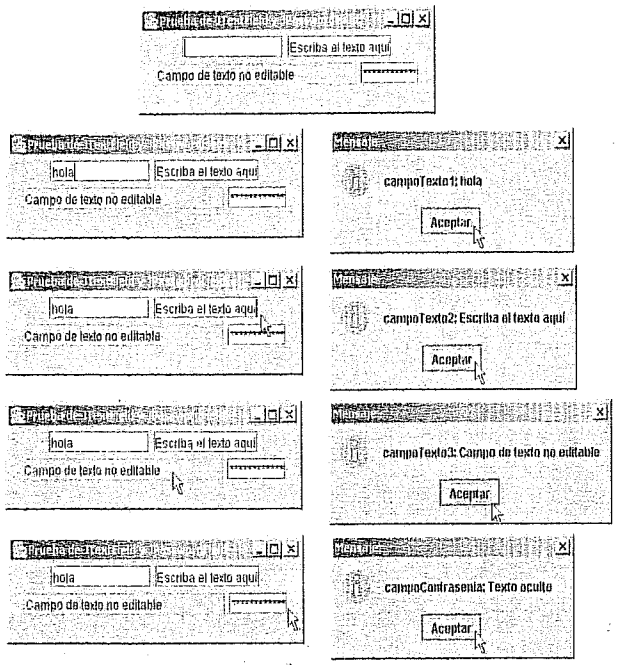


Figura 13.7 Objetos `JTextField` y `JPasswordField`. (Parte 3 de 3.)

`Focus` de ese componente en el programa. En este ejemplo, al ocurrir un evento en el objeto `JPasswordField`, la contraseña se revela.

En la línea 8 se declaran tres variables `JTextField` (`campoTexto1`, `campoTexto2` y `campoTexto3`), y en la línea 9 se declara una variable `JPasswordField` (`campoContrasenia`). Cada uno de los campos de texto correspondientes se instancian en el constructor (líneas 12 a 47). En la línea 20 se crea `campoTexto1` con 10 columnas de texto. El tamaño en píxeles de una columna de texto se determina en base a la anchura promedio de un carácter en el tipo de letra actual del campo de texto. En la línea 21 se agrega `campoTexto1` al panel de contenido.

En la línea 24 se crea `campoTexto2` con el texto inicial `"Escriba el texto aquí"` a mostrar en el campo de texto. La anchura del campo de texto se determina en base a la anchura del texto predeterminado. En la línea 25 se agrega `campoTexto2` al panel de contenido.

En la línea 29 se crea `campoTexto3` y se llama al constructor de `JTextField` con dos argumentos: el texto predeterminado `"Campo de texto no editable"` a mostrar y el número de columnas especificadas. En la línea 30 se utiliza el método *`setEditable`* (heredado en `JTextField` de la clase `JTextCompo-`

`nent`) para hacer que el campo de texto no pueda editarse; es decir, el usuario no podrá modificar el texto en este campo de texto. En la línea 31 se agrega `campoTexto3` al panel de contenido.

En la línea 34 se crea `campoContrasenia` con el texto `"Texto oculto"` a mostrar en el campo de texto. La anchura de este campo de texto se determina en base a la anchura del texto predeterminado. Al ejecutar el programa, observe que el texto se muestra como una cadena de asteriscos. En la línea 35 se agrega `campoContrasenia` al panel de contenido.

Para el manejo de eventos en este ejemplo, hemos declarado la clase interna `ManejadorCampoTexto` (líneas 56 a 85), la cual implementa a la interfaz `ActionListener`. (Hablaremos sobre la clase `ManejadorCampoTexto` en breve.) Por lo tanto, toda instancia de la clase `ManejadorCampoTexto` *es un* `ActionListener`. En la línea 38 se crea una instancia de la clase `ManejadorCampoTexto` y se asigna su referencia a `manejador`. Esta única instancia se utilizará como el objeto de escucha de eventos para los tres objetos `JTextField` y el objeto `JPasswordField` en este ejemplo.

En las líneas 39 a 42 están las instrucciones de registro de eventos que especifican el objeto de escucha de eventos para cada componente de la GUI. El programa llama al método `addActionListener` de `JTextField` para registrar el evento para cada componente. El método `addActionListener` recibe como su argumento un objeto `ActionListener`. Por lo tanto, cualquier objeto de una clase que implemente a la interfaz `ActionListener` (es decir, cualquier objeto que *sea un* `ActionListener`) podrá suministrarse como argumento para este método. El objeto al que `manejador` hace referencia *es un* `ActionListener`, ya que su clase implementa a la interfaz `ActionListener`. Una vez que se ejecutan estas instrucciones, el objeto al que `manejador` hace referencia *escucha los eventos* (es decir, recibirá una notificación cuando ocurra un evento) en estos cuatro objetos. Ahora, cuando el usuario oprima *Intro* en cualquiera de estos cuatro campos, el método `actionPerformed` (líneas 59 a 83) de la clase `ManejadorCampoTexto` será llamado para manejar el evento.

Observación de ingeniería de software 13.2



El componente de escucha de eventos para cierto evento debe implementar a la interfaz de escucha de eventos apropiada.

El método `actionPerformed` utiliza el método `getSource` de su argumento `ActionEvent` (líneas 64, 68, 72 y 76) para determinar el componente de la GUI con el cual interactuó el usuario y crea una cadena para mostrarla en un cuadro de diálogo de mensaje. El método `getActionCommand` de `ActionEvent` devuelve el texto en el objeto `JTextField` que generó el evento. Si el usuario interactuó con el objeto `JPasswordField`, en las líneas 77 y 78 se usa el método *`getPassword`* de `JPasswordField` para obtener la contraseña y crear la cadena a mostrar. El método `getPassword` devuelve la contraseña como un arreglo de tipo `char`, que se utiliza como argumento para un constructor de `String`, para crear una cadena que contiene los caracteres del arreglo. En la línea 81 se muestra un cuadro de mensaje indicando el nombre de la referencia al componente de la GUI y el texto que el usuario escribió en el campo.

Incluso hasta un objeto `JTextField` no editable puede generar un evento; simplemente haga clic en el campo de texto y oprima *Intro*. Además, observe que el texto actual de la contraseña se muestra cuando oprime *Intro* en el objeto `JPasswordField`. (¡Desde luego que usted, normalmente, no haría esto!)

Error común de programación 13.3



Olvidar registrar un objeto manejador de eventos para un tipo de evento específico de un componente de la GUI hace que los eventos de ese tipo se ignoren.

Como dijimos en la sección 13.4, el componente de la GUI con el cuál interactúa el usuario es el origen del evento. En este ejemplo, el origen del evento es uno de los campos de texto o el campo de contraseña. Cuando el usuario oprime *Intro* mientras uno de esos campos tiene el enfoque, se crea un objeto `ActionEvent` único. Este objeto se pasa después, mediante una llamada a un método, al componente de escucha de eventos (declarado en las líneas 56 a 85), que se encarga de manejar el evento.

13.6 Cómo funciona el manejo de eventos

Mostraremos cómo funciona el mecanismo de manejo de eventos, utilizando a `campoTexto1` del ejemplo de la figura 13.7. Tenemos dos preguntas sin contestar de la sección 13.4:

http://librosysolucionarios.net

1. ¿Cómo se registró el manejador de eventos?
2. ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a algún otro método manejador de eventos?

La primera pregunta se responde mediante el registro de eventos que se lleva a cabo en las líneas 39 a 42 del programa. En la figura 13.8 se muestra un diagrama de la variable `JTextField` llamada `campoTexto1`, el objeto `JTextField` al que hace referencia y el objeto de escucha que se registra para manejar el evento del objeto `JTextField`.

Todo `JComponent` tiene un objeto de la clase `EventListenerList` (paquete `javax.swing.event`) llamado `listenerList`, como una variable de instancia. Las referencias a todos los componentes de escucha registrados se almacenan en la variable `listenerList` (que se muestra en el diagrama como una flecha debajo del objeto `JTextField` en la figura 13.8).

Cuando se ejecuta la instrucción:

```
campoTexto1.addActionListener( manejador );
```

en la figura 13.7, se coloca una nueva entrada en la variable `listenerList` de `campoTexto1`, indicando tanto la referencia al objeto de escucha como el tipo de componente de escucha (en este caso, `ActionListener`). Por lo tanto, todo objeto de un componente de la GUI tiene referencias a todos sus componentes de escucha registrados.

El tipo de componente de escucha de eventos es importante para responder a la segunda pregunta: ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a algún otro método manejador de eventos? Todo componente de la GUI soporta varios tipos de eventos, incluyendo *eventos de ratón*, *eventos de tecla* y otros más. Cuando ocurre un evento, éste se *despacha* solamente a los componentes de escucha de eventos del tipo apropiado. El despachamiento (dispatching) es simplemente el proceso por el cual se llama al método manejador de eventos apropiado en cada componente de escucha registrado, para el tipo de evento que ocurrió.

Cada tipo de evento tiene su correspondiente interfaz de escucha de eventos. Por ejemplo, los eventos tipo `ActionEvent` son manejados por objetos `ActionListener`, los eventos tipo `MouseEvent` son manejados por objetos `MouseListener` (y `MouseMotionListener`, como veremos en breve) y los eventos tipo `KeyEvent` son manejados por objetos `KeyListener`. Cuando ocurre un evento, el componente de la GUI recibe un *ID de evento* único, el cual especifica el tipo de evento. El componente de la GUI utiliza el ID de evento para decidir a cuál tipo de componente de escucha debe despacharse el evento, y para decidir cuál método llamar en cada objeto de escucha. Para un `ActionEvent`, el evento se despacha al método `actionPerformed` de todos los objetos `ActionListener` registrados (el único método en la interfaz `ActionListener`). En el caso de un `MouseEvent`, el evento se despacha a todos los objetos `MouseListener` registrados (o `MouseMotionListener`, dependiendo del evento de ratón que ocurra). El ID de evento del objeto `MouseEvent` determina cuáles de los varios métodos manejadores de eventos de ratón son llamados.

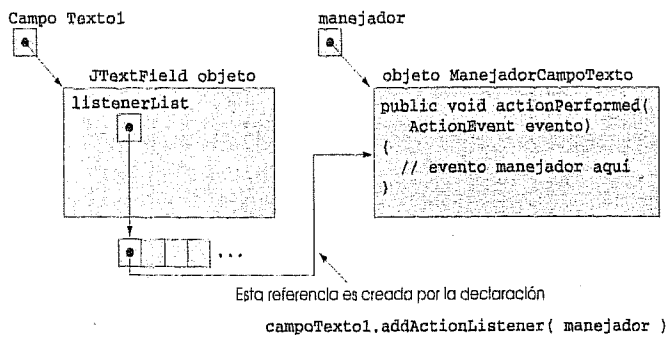


Figura 13.8 Registro de eventos para `JTextField` `campoTexto1`.

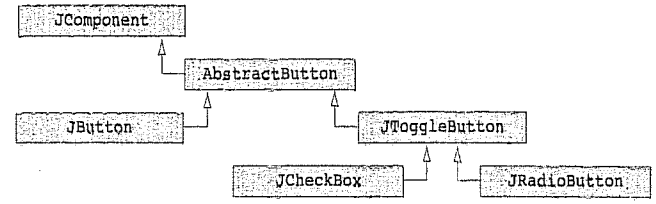


Figura 13.9 Jerarquía de botones de Swing.

Toda esta lógica de decisión la administran los componentes de la GUI por usted. Hablaremos sobre otros tipos de eventos e interfaces de escucha de eventos a medida que se vayan necesitando con cada nuevo componente que vayamos viendo.

13.7 `JButton`

Un *botón* es un componente en el que el usuario hace clic para desencadenar cierta acción. Un programa de Java puede utilizar varios tipos de botones, incluyendo *botones de comando*, *casillas de verificación*, *botones interruptores* y *botones de opción*. En la figura 13.9 se muestra la jerarquía de herencia de los botones de Swing que veremos en este capítulo. Como puede ver en el diagrama, todos los tipos de botones son subclases de `AbstractButton` (paquete `javax.swing`), la cual declara las características comunes para los botones de Swing. En esta sección nos concentraremos en los botones que se utilizan comúnmente para iniciar un comando.

Un botón de comando genera un evento `ActionEvent` cuando el usuario hace clic en él. Los botones de comando se crean con la clase `JButton` (que presentamos en la sección 6.8), la cual extiende a la clase `AbstractButton`. El texto en la cara de un objeto `JButton` se llama *etiqueta del botón*. Una GUI puede tener muchos objetos `JButton`, pero cada etiqueta de botón debe generalmente ser única en las partes de la GUI en que se muestre.

Observación de apariencia visual 13.4

Tener más de un objeto `JButton` con la misma etiqueta hace que los objetos `JButton` sean ambiguos para el usuario. Debe proporcionar una etiqueta única para cada botón.

La aplicación de la figura 13.10 crea dos objetos `JButton` y demuestra que estos objetos, al igual que los objetos `JLabel`, tienen soporte para mostrar objetos `Icon`. El manejo de eventos para los botones se lleva a cabo mediante una sola instancia de la clase interna `ManejadorBoton` (declarado en las líneas 47 a 56).

```
1  // Fig. 13.10: PruebaBoton.java
2  // Creación de objetos JButton.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaBoton extends JFrame {
8     private JButton botonSimple, botonElegante;
9
10    // configurar la GUI
11    public PruebaBoton()
12    {
13       super( "Prueba de botones" );
14
15       // obtener panel de contenido y establecer su esquema
```

Figura 13.10 Botones de comando y eventos de acción. (Parte 1 de 3.)

http://librosysolucionarios.net

```
16        Container contenedor = getContentPane();
17        contenedor.setLayout( new FlowLayout() );
18
19        // crear botones
20        botonSimple = new JButton( "Botón simple" );
21        contenedor.add( botonSimple );
22
23        Icon insecto1 = new ImageIcon( "insecto1.gif" );
24        Icon insecto2 = new ImageIcon( "insecto2.gif" );
25        botonElegante = new JButton( "Botón elegante", insecto1 );
26        botonElegante.setRolloverIcon( insecto2 );
27        contenedor.add( botonElegante );
28
29        // crear una instancia de la clase interna ManejadorBoton
30        // a usar para el manejo de eventos de botones
31        ManejadorBoton manejador = new ManejadorBoton();
32        botonElegante.addActionListener( manejador );
33        botonSimple.addActionListener( manejador );
34
35        setSize( 275, 100 );
36        setVisible( true );
37
38     } // fin del constructor de PruebaBoton
39
40     public static void main( String args[] )
41     {
42        PruebaBoton aplicacion = new PruebaBoton();
43        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
44     }
45
46     // clase interna para el manejo de eventos de botón
47     private class ManejadorBoton implements ActionListener {
48
49        // manejar evento de botón
50        public void actionPerformed( ActionEvent evento )
51        {
52           JOptionPane.showMessageDialog( PruebaBoton.this,
53              "Usted oprimió: " + evento.getActionCommand() );
54        }
55
56     } // fin de la clase interna privada ManejadorBoton
57
58  } // fin de la clase PruebaBoton
```

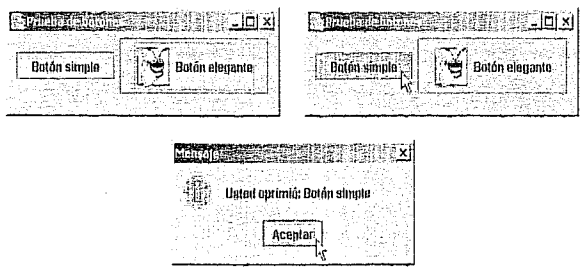


Figura 13.10 Botones de comando y eventos de acción. (Parte 2 de 3.)

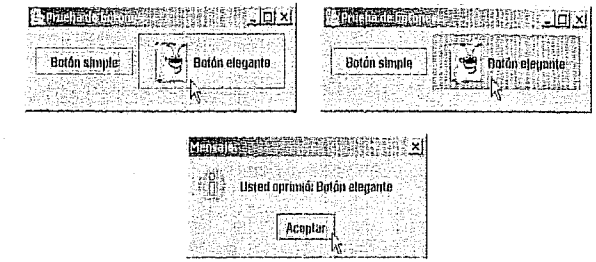


Figura 13.10 Botones de comando y eventos de acción. (Parte 3 de 3.)

En la línea 8 se declaran las variables `botonSimple` y `botonElegante` de la clase `JButton`. Los correspondientes objetos se instancian en el constructor. En la línea 20 se crea `botonSimple` con la etiqueta `"Boton simple"`. En la línea 21 se agrega el botón al panel de contenido.

Un objeto `JButton` puede mostrar un objeto `Icon`. Para proveer al usuario un nivel adicional de interacción visual con la GUI, un objeto `JButton` puede tener también un objeto `Icon` *de sustitución*: un `Icon` que se muestre cuando el usuario coloque el ratón encima del botón. El icono en el botón cambia a medida que el ratón se mueve hacia dentro y fuera del área del botón en la pantalla. En las líneas 23 y 24 se crean dos objetos `ImageIcon` que representan al objeto `Icon` predeterminado y el objeto `Icon` de sustitución para el objeto `JButton` creado en la línea 25. Ambas instrucciones suponen que los archivos de imagen están guardados en el mismo directorio que el programa (que es comúnmente el caso para las aplicaciones que utilizan imágenes).

En la línea 25 se crea `botonElegante` con el texto `"Boton elegante"` y el icono `insecto1`. De manera predeterminada, el texto se muestra a la derecha del icono. En la línea 26 se utiliza el método *`setRolloverIcon`* (heredado de `AbstractButton`) para especificar la imagen a mostrar en el botón cuando el usuario coloque el ratón sobre el botón. En la línea 27 se agrega el botón al panel de contenido.

Observación de apariencia visual 13.5

Como la clase `AbstractButton` soporta el mostrar texto e imágenes en un botón, todas las subclases de `AbstractButton` soportan también el mostrar texto e imágenes.

Observación de apariencia visual 13.6

Al usar iconos de sustitución para los objetos `JButton`, los usuarios reciben una retroalimentación visual que les indica que, al hacer clic en el ratón mientras el cursor está colocado encima del botón, ocurrirá una acción.

Los objetos `JButton`, al igual que los objetos `JTextField`, generan eventos `ActionEvent`. Como lo mencionamos anteriormente, un evento `ActionEvent` puede ser procesado por cualquier objeto `ActionListener`. En las líneas 31 a 33 se crea un objeto de escucha de eventos y se registra un objeto `ActionListener` para cada objeto `JButton`. La clase interna `ManejadorBoton` (líneas 47 a 56) declara a `actionPerformed` para mostrar un cuadro de diálogo de mensaje que contiene la etiqueta del botón que el usuario oprimió. Para un evento de `JButton`, el método `getActionCommand` de `ActionEvent` devuelve la etiqueta del botón.

13.8 `JCheckBox` y `JRadioButton`

Los componentes de la GUI de Swing contienen tres tipos de *botones de estado*: *`JToggleButton`*, *`JCheckBox`* y *`JRadioButton`*, los cuales tienen valores encendido/apagado o verdadero/falso. Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton` (figura 13.9). Un objeto `JRadioButton` es distinto de un objeto `JCheckBox` en cuanto a que generalmente hay varios objetos `JRadioButton` que se agrupan, y sólo uno de los objetos `JRadioButton` en el grupo puede estar seleccionado (verdadero) en un momento dado. Primero veremos la clase `JCheckBox`.

http://librosysolucionarios.net

JCheckBox

En la figura 13.11 se utilizan dos objetos **JCheckBox** para seleccionar el estilo deseado de tipo de letra para el texto a mostrar en un objeto **JTextField**. Un objeto **JCheckBox** aplica un estilo en negritas cuando se selecciona, y el otro aplica un estilo en cursivas al ser seleccionado. Si ambos se seleccionan, el estilo del tipo de letra es negrita y cursiva. Cuando el programa se ejecuta por primera vez, ninguno de los objetos **JCheckBox** está activado (es decir, ambos son **false**), por lo que el tipo de letra es simple.

Una vez creado e inicializado el objeto **JTextField**, en la línea 22 se establece el tipo de letra de este objeto en estilo **Serif**, **PLAIN** y tamaño de 14 puntos. A continuación, en las líneas 26 y 29 se crean dos objetos **JCheckBox**. La cadena que se pasa al constructor de **JCheckBox** es la *etiqueta de la casilla de verificación* que aparece a la derecha del objeto **JCheckBox** de manera predeterminada.

```
1  // Fig. 13.11: PruebaCasillaVerificacion.java
2  // Creación de botones JCheckBox.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaCasillaVerificacion extends JFrame {
8     private JTextField campo;
9     private JCheckBox negrita, cursiva;
10
11    // configurar la GUI
12    public PruebaCasillaVerificacion()
13    {
14       super( "Prueba de JCheckBox" );
15
16       // obtener panel de contenido y establecer su esquema
17       Container contenedor = getContentPane();
18       contenedor.setLayout( new FlowLayout() );
19
20       // configurar objeto JTextField y establecer su tipo de letra
21       campo = new JTextField( "Observe el cambio en el estilo de tipo de letra", 25 );
22       campo.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
23       contenedor.add( campo );
24
25       // crear objetos casilla de verificación
26       negrita = new JCheckBox( "Negrita" );
27       contenedor.add( negrita );
28
29       cursiva = new JCheckBox( "Cursiva" );
30       contenedor.add( cursiva );
31
32       // registrar componentes de escucha para los objetos JCheckBox
33       ManejadorCasillaVerificacion manejador = new ManejadorCasillaVerificacion();
34       negrita.addItemListener( manejador );
35       cursiva.addItemListener( manejador );
36
37       setSize( 300, 100 );
38       setVisible( true );
39
40    } // fin del constructor de PruebaCasillaVerificacion
41
42    public static void main( String args[] )
```

Figura 13.11 Botones **JCheckBox** y eventos de los elementos. (Parte 1 de 2.)


```
43    {
44       PruebaCasillaVerificacion aplicacion = new PruebaCasillaVerificacion();
45       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
46    }
47
48    // clase interna privada para manejo de eventos de ItemListener
49    private class ManejadorCasillaVerificacion implements ItemListener {
50       private int valNegrita = Font.PLAIN;
51       private int valCursiva = Font.PLAIN;
52
53       // responder a eventos de casilla de verificación
54       public void itemStateChanged( ItemEvent evento )
55       {
56          // procesar eventos de casilla de verificación negrita
57          if ( evento.getSource() == negrita )
58             valNegrita = negrita.isSelected() ? Font.BOLD : Font.PLAIN;
59
60          // procesar eventos de casilla de verificación cursiva
61          if ( evento.getSource() == cursiva )
62             valCursiva = cursiva.isSelected() ? Font.ITALIC : Font.PLAIN;
63
64          // establecer tipo de letra del campo de texto
65          campo.setFont( new Font( "Serif", valNegrita + valCursiva, 14 ) );
66
67       } // fin del método itemStateChanged
68
69    } // fin de la clase interna privada ManejadorCasillaVerificacion
70
71 } // fin de la clase PruebaCasillaVerificacion
```

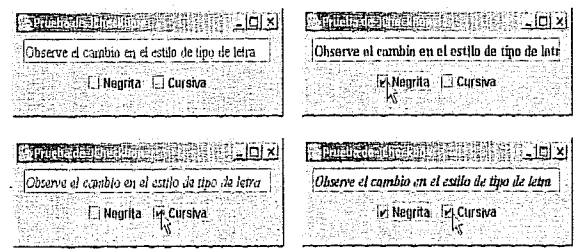


Figura 13.11 Botones **JCheckBox** y eventos de los elementos. (Parte 2 de 2.)

Cuando el usuario hace clic en un objeto **JCheckBox**, ocurre un evento **ItemEvent** que puede manejarse mediante un **ItemListener** (cualquier objeto de una clase que implemente a la interfaz **ItemListener**). Un **ItemListener** debe implementar el método ***itemStateChanged***. En este ejemplo, el manejo de eventos se lleva a cabo mediante una instancia de la clase interna **ManejadorCasillaVerificacion** (líneas 49 a 69). En las líneas 33 a 35 se crea una instancia de la clase **ManejadorCasillaVerificacion** y se registra con el método ***addItemListener*** como componente de escucha para las casillas de verificación **negrita** y **cursiva**.

El método **itemStateChanged** (líneas 54 a 67) es llamado cuando el usuario hace clic en la casilla de verificación **negrita** o **cursiva**. Este método utiliza **evento.getSource()** para determinar en cuál de los objetos **CheckBox** se hizo clic. Si fue en la casilla de verificación **negrita**, en la línea 58 se utiliza el método ***isSelected*** de **JCheckBox** para determinar si el botón está seleccionado. Si es así, a la variable local **valNegrita** se le asigna **Font.BOLD**; en caso contrario, se le asigna **Font.PLAIN**. Una instrucción simi-

http://librosysolucionarios.net

lar se ejecuta si se hace clic en **cursiva**. Si esta casilla de verificación está seleccionada, a la variable local **valCursiva** se le asigna **Font.ITALIC**; en caso contrario, se le asigna **Font.PLAIN**. En la línea 65 se usa la suma de **valBold** y **valItalic** como el estilo del nuevo tipo de letra del campo de texto.

JRadioButton

Los *botones de opción* (que se declaran con la clase ***JRadioButton***) son similares a las casillas de verificación, en cuanto a que tienen dos estados: *seleccionado* y *no seleccionado* (al que también se le conoce como *deseleccionado*). Sin embargo, los botones de opción generalmente aparecen como un *grupo*, en el cual sólo un botón de opción puede estar seleccionado en un momento dado. Al seleccionar un botón de opción distinto en el grupo se obliga a que todos los demás botones de opción del grupo se deseleccionen. Los botones de opción se utilizan para representar un conjunto de opciones *mutuamente exclusivas* (es decir, no pueden seleccionarse varias opciones en el grupo al mismo tiempo). La relación lógica entre los botones de opción se mantiene mediante un objeto ***ButtonGroup*** (paquete **javax.swing**), el cual en sí no es un componente de la GUI. Por lo tanto, un objeto **ButtonGroup** organiza un grupo de botones y no se muestra a sí mismo en una interfaz de usuario. En vez de ello, se muestra en la GUI cada uno de los objetos **JRadioButton**.



Error común de programación 13.4

Si se agrega un objeto **ButtonGroup** *(o un objeto de cualquier otra clase que no se derive de* **Component***) a un contenedor se produce un error de compilación.*

La aplicación de la figura 13.12 es similar a la de la figura 13.11. El usuario puede alterar el estilo del tipo de letra del texto de un objeto **JTextField**. El programa utiliza botones de opción que permiten que se seleccione solamente un estilo de tipo de letra en el grupo a la vez.

En las líneas 28 a 38 del constructor se crean cuatro objetos **JRadioButton** y se agregan al panel de contenido de la ventana de aplicación. Cada objeto **JRadioButton** se crea con una llamada al constructor como la de la línea 28. Este constructor inicializa la etiqueta que aparece a la derecha del objeto **JRadioButton** de manera predeterminada, junto con su estado inicial. Un segundo argumento **true** indica que el objeto **JRadioButton** debe aparecer seleccionado al mostrarlo en pantalla.

```
1  // Fig. 13.12: PruebaBotonOpcion.java
2  // Creación de botones de opción, utilizando ButtonGroup y JRadioButton.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaBotonOpcion extends JFrame {
8    private JTextField campo;
9    private Font tipoLetraSimple, tipoLetraNegrita, tipoLetraCursiva, tipoLetraNegritaCursiva;
10    private JRadioButton botonSimple, botonNegrita, botonCursiva,
11       botonNegritaCursiva;
12    private ButtonGroup grupoBotonesOpcion;
13
14    // crear la GUI y tipos de letra
15    public PruebaBotonOpcion()
16    {
17       super( "Prueba de RadioButton" );
18
19       // obtener panel de contenido y establecer su esquema
20       Container contenedor = getContentPane();
21       contenedor.setLayout( new FlowLayout() );
22
23       // establecer JTextField
24       campo = new JTextField( "Observe cómo cambia el estilo del tipo de letra", 30 );
25       contenedor.add( campo );
```



```
26
27       // crear botones de opción
28       botonSimple = new JRadioButton( "Simple", true );
29       contenedor.add( botonSimple );
30
31       botonNegrita = new JRadioButton( "Negrita", false );
32       contenedor.add( botonNegrita );
33
34       botonCursiva = new JRadioButton( "Cursiva", false );
35       contenedor.add( botonCursiva );
36
37       botonNegritaCursiva = new JRadioButton( "Negrita/Cursiva", false );
38       contenedor.add( botonNegritaCursiva );
39
40       // crear relación lógica entre objetos JRadioButton
41       grupoBotonesOpcion = new ButtonGroup();
42       grupoBotonesOpcion.add( botonSimple );
43       grupoBotonesOpcion.add( botonNegrita );
44       grupoBotonesOpcion.add( botonCursiva );
45       grupoBotonesOpcion.add( botonNegritaCursiva );
46
47       // crear objetos de tipo de letra
48       tipoLetraSimple = new Font( "Serif", Font.PLAIN, 14 );
49       tipoLetraNegrita = new Font( "Serif", Font.BOLD, 14 );
50       tipoLetraCursiva = new Font( "Serif", Font.ITALIC, 14 );
51       tipoLetraNegritaCursiva = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
52       campo.setFont( tipoLetraSimple );  // establecer tipo de letra inicial
53
54       // registrar eventos para objetos JRadioButton
55       botonSimple.addItemListener( new ManejadorBotonOpcion( tipoLetraSimple ) );
56       botonNegrita.addItemListener( new ManejadorBotonOpcion( tipoLetraNegrita ) );
57       botonCursiva.addItemListener(
58          new ManejadorBotonOpcion( tipoLetraCursiva ) );
59       botonNegritaCursiva.addItemListener(
60          new ManejadorBotonOpcion( tipoLetraNegritaCursiva ) );
61
62       setSize( 350, 100 );
63       setVisible( true );
64
65    } // fin del constructor de PruebaBotonOpcion
66
67    public static void main( String args[] )
68    {
69       PruebaBotonOpcion aplicacion = new PruebaBotonOpcion();
70       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
71    }
72
73    // clase interna privada para manejar eventos de botón de opción
74    private class ManejadorBotonOpcion implements ItemListener {
75       private Font tipoDeLetra;
76
77       public ManejadorBotonOpcion( Font f )
78       {
79          tipoDeLetra = f;
```

http://librosysolucionarios.net

```
80         }
81
82         // manejar eventos de botón de opción
83         public void itemStateChanged( ItemEvent evento )
84         {
85            campo.setFont( tipoDeLetra );
86         }
87
88      } // fin de la clase interna privada ManejadorBotonOpcion
89
90   } // fin de la clase PruebaBotonOpcion
```

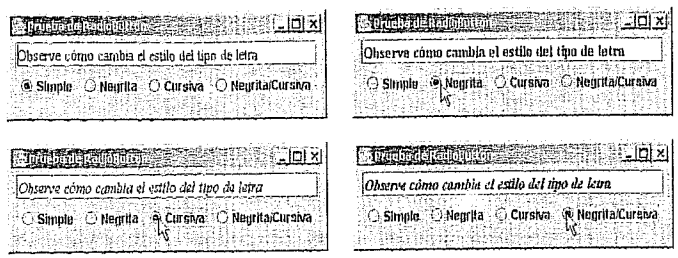


Figura 13.12 Objetos `JRadioButton` y `ButtonGroup`. (Parte 3 de 3.)

Los objetos `JRadioButton`, al igual que los objetos `JCheckBox`, generan eventos tipo `ItemEvent` cuando se hace clic sobre ellos. En las líneas 55 a 60 se crean instancias de la clase interna `ManejadorBotonOpcion` (declarada en las líneas 84 a 88) y se registra cada una de esas instancias para manejar un evento `ItemEvent` que se genere cuando el usuario hace clic en cualquiera de los objetos `JRadioButton`.

En la línea 41 se instancia un objeto `ButtonGroup` y se asigna a la referencia `grupoBotonesOpcion`. Este objeto es el "pegamento" que forma la relación lógica entre los cuatro objetos `JRadioButton` y permite que se seleccione solamente uno de los cuatro en un momento dado. En las líneas 42 a 45 se utiliza el método *add* de `ButtonGroup` para asociar cada uno de los objetos `JRadioButton` con `grupoBotonesOpcion`. Si se agrega al grupo más de un objeto `JRadioButton` seleccionado, el primer objeto `JRadioButton` seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.

La clase `ManejadorBotonOpcion` (línea 74 a 88) implementa la interfaz `ItemListener` para poder manejar los eventos de elementos generados por los objetos `JRadioButton`. Cada objeto `JRadioButton` en el programa tiene registrada una instancia distinta de esta clase como su objeto `ItemListener`. Al crearse estas instancias, se pasa un objeto `Font` en cada llamada al constructor. El constructor almacena este tipo de letra en una variable de instancia (línea 79) en el objeto manejador de eventos. Cuando el usuario hace clic en un objeto `JRadioButton`, `grupoBotonesOpcion` desactiva el objeto `JRadioButton` previamente seleccionado y el método `itemStateChanged` (líneas 83 a 86) establece el tipo de letra en el objeto `JTextField` al tipo de letra almacenado en el objeto manejador de eventos.

13.9 JComboBox

Un *cuadro combinado* (algunas veces conocido como *lista desplegable*) proporciona una lista de elementos, de la cual el usuario puede seleccionar solamente uno. Los cuadros combinados se implementan con la clase *JComboBox*, la cual extiende a la clase `JComponent`. Los objetos `JComboBox` generan eventos `ItemEvent`, al igual que los objetos `JCheckBox` y `JRadioButton`.

La aplicación de la figura 13.13 utiliza un objeto `JComboBox` para proporcionar una lista de cuatro nombres de archivos de imágenes. Cuando se selecciona el nombre de un archivo de imagen, la imagen correspondiente se muestra como un objeto `Icon` en un objeto `JLabel`. Las capturas de pantalla para este programa

http://librosysolucionarios.net

muestran la lista `JComboBox` después de hacer una selección, para ilustrar cuál nombre de archivo de imagen fue seleccionado.

En las líneas 13 a 15 se declara e inicializa el arreglo `iconos` con cuatro nuevos objetos `ImageIcon`. El arreglo `String` llamado `nombres` (declarado en las líneas 11 y 12) contiene los nombres de los cuatro archivos de imagen que se guardan en el mismo directorio que la aplicación.

En la línea 27 se crea un objeto `JComboBox`, utilizando los objetos `String` en el arreglo `nombres` como los elementos en la lista. Cada elemento de la lista tiene un *índice*. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado al mostrar el objeto `JComboBox`. Los otros elementos se seleccionan haciendo clic en el objeto `JComboBox`. Cuando se hace clic en este objeto, se expande en una lista de la cual el usuario puede hacer una selección.

En la línea 28 se utiliza el método `setMaximumRowCount` del método `JComboBox` para establecer el máximo número de elementos a mostrar en 3, cuando el usuario haga clic en el objeto `JComboBox`. Si hay más elementos en el objeto `JComboBox` que el número máximo de elementos que se muestran en pantalla, el objeto `JComboBox` proporciona una *barra de desplazamiento* (vea la primera captura de pantalla) que permite al usuario desplazarse por todos los elementos en la lista. El usuario puede hacer clic en las *flechas de desplazamiento* que están en las partes superior e inferior de la barra de desplazamiento para avanzar hacia arriba y hacia abajo de la lista, un elemento a la vez, o puede arrastrar hacia arriba y hacia abajo el *cuadro de desplazamiento* que está en medio de la barra de desplazamiento para desplazarse por la lista. Para arrastrar el cuadro de desplazamiento, mantenga presionado el botón izquierdo del ratón mientras éste se encuentra sobre el cuadro de desplazamiento, y mueva el ratón.

```
1  // Fig. 13.13: PruebaCuadroComb.java
2  // Uso de un objeto JComboBox para seleccionar una imagen a mostrar.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaCuadroComb extends JFrame {
8     private JComboBox cuadroCombImagenes;
9     private JLabel etiqueta;
10
11    private String nombres[] =
12       { "insecto1.gif", "insecto2.gif",  "insectoviaje.gif", "insectanim.gif" };
13    private Icon iconos[] = { new ImageIcon( nombres[ 0 ] ),
14       new ImageIcon( nombres[ 1 ] ), new ImageIcon( nombres[ 2 ] ),
15       new ImageIcon( nombres[ 3 ] ) };
16
17    // configurar la GUI
18    public PruebaCuadroComb()
19    {
20       super( "Prueba de JComboBox" );
21
22       // obtener panel de contenido y establecer su esquema
23       Container contenedor = getContentPane();
24       contenedor.setLayout( new FlowLayout() );
25
26       // establecer objeto JComboBox y registrar su manejador de eventos
27       cuadroCombImagenes = new JComboBox( nombres );
28       cuadroCombImagenes.setMaximumRowCount( 3 );
29       cuadroCombImagenes.addItemListener(
30
31          new ItemListener() {  // clase interna anónima
```

Figura 13.13 Objeto `JComboBox` que muestra una lista de nombres de imágenes. (Parte 1 de 2.)

```
32
33             // manejar evento para JComboBox
34             public void itemStateChanged( ItemEvent evento )
35             {
36                // determinar si la casilla de verificación está seleccionada
37             if ( evento.getStateChange() == ItemEvent.SELECTED )
38                   etiqueta.setIcon( iconos[
39                      cuadroCombImagenes.getSelectedIndex() ] );
40             }
41
42          } // fin de la clase interna anónima
43
44       ); // fin de la llamada a addItemListener
45
46       contenedor.add( cuadroCombImagenes );
47
48       // establecer objeto JLabel para mostrar objetos ImageIcon
49       etiqueta = new JLabel( iconos[ 0 ] );
50       contenedor.add( etiqueta );
51
52       setSize( 350, 100 );
53       setVisible( true );
54
55    } // fin del constructor de PruebaCuadroComb
56
57    public static void main( String args[] )
58    {
59       PruebaCuadroComb aplicacion = new PruebaCuadroComb();
60       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
61    }
62
63 } // fin de la clase PruebaCuadroComb
```

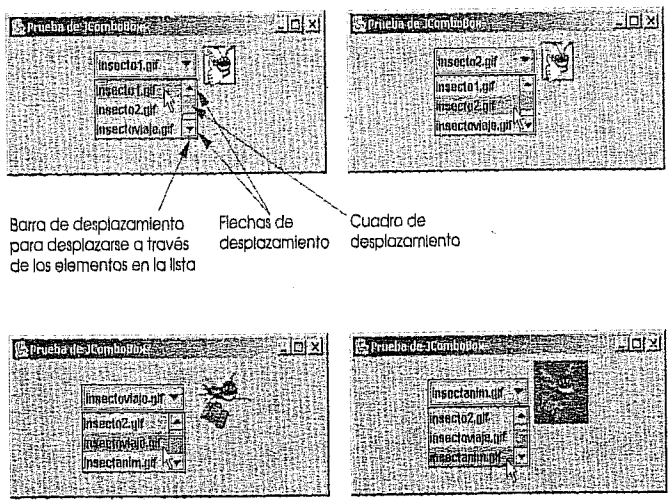


Figura 13.13 Objeto `JComboBox` que muestra una lista de nombres de imágenes. (Parte 2 de 2.)

http://librosysolucionarios.net

Observación de apariencia visual 13.7

Establezca el número máximo de filas en un objeto `JComboBox` a un valor que evite que la lista se expanda fuera de los límites de la ventana o subprograma en la que se utilice. Esta configuración asegurará que la lista se muestre correctamente cuando el usuario la expanda.

En las líneas 29 a 44 se registra una instancia de una clase interna anónima que implementa a `ItemListener` como la interfaz de escucha para el objeto `JComboBox` llamado `imagenes`. Cuando el usuario selecciona un elemento de `imagenes`, el método `itemStateChanged` (líneas 34 a 40) establece el objeto `Icon` para `etiqueta`. Este objeto se selecciona del arreglo `iconos`, determinando el índice del elemento seleccionado en el objeto `JComboBox` con el método `getSelectedIndex` en la línea 39. Observe que en la línea 37 se cambia el icono sólo para un elemento seleccionado. El método `getStateChange` determina el estado de un objeto `JToggleButton`. La razón para que haya una instrucción `if` aquí es que, para cada elemento que sea seleccionado de un objeto `JComboBox`, se deselecciona otro elemento. Por lo tanto, ocurren dos eventos por cada elemento seleccionado. Deseamos mostrar solamente el icono para el elemento que el usuario acaba de seleccionar.

13.10 `JList`

Una *lista* muestra una serie de elementos, de la cual el usuario puede seleccionar uno o más. Las listas se crean con la clase `JList`, que extiende directamente a la clase `JComponent`. La clase `JList` soporta *listas de selección simple* (listas que permiten seleccionar solamente un elemento a la vez) y *listas de selección múltiple* (listas que permiten seleccionar cualquier número de elementos a la vez). En esta sección hablaremos sobre las listas de selección simple.

La aplicación de la figura 13.14 crea un objeto `JList` que contiene los nombres de 13 colores. Al hacer clic en el nombre de un color en el objeto `JList`, ocurre un evento `ListSelectionEvent` y el programa cambia el color de fondo de la ventana de aplicación al color seleccionado.

```
1  // Fig. 13.14: PruebaLista.java
2  // Selección de colores de un objeto JList.
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.event.*;
6
7  public class PruebaLista extends JFrame {
8     private JList listaColores;
9     private Container contenedor;
10
11    private final String nombresColores[] = { "Negro", "Azul", "Cyan",
12       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta",
13       "Naranja", "Rosa", "Rojo", "Blanco", "Amarillo" };
14
15    private final Color colores[] = { Color.BLACK, Color.BLUE, Color.CYAN,
16       Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
17       Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
18       Color.YELLOW };
19
20    // configurar la GUI
21    public PruebaLista()
22    {
23       super( "Prueba de JList" );
24
25       // obtener panel de contenido y establecer su esquema
26       contenedor = getContentPane();
```

Figura 13.14 Objeto `JList` que muestra una lista de colores. (Parte 1 de 2.)

```
27        contenedor.setLayout( new FlowLayout() );
28
29        // crear una lista con elementos del arreglo nombresColores
30        listaColores = new JList( nombresColores );
31        listaColores.setVisibleRowCount( 5 );
32
33        // no permitir selecciones múltiples
34        listaColores.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
35
36        // agregar un objeto JScrollPane, que contiene a JList, al panel de contenido
37        contenedor.add( new JScrollPane( listaColores ) );
38        listaColores.addListSelectionListener(
39
40           new ListSelectionListener() {  // clase interna anónima
41
42              // manejar eventos de selección en la lista
43              public void valueChanged( ListSelectionEvent evento )
44              {
45                 contenedor.setBackground(
46                    colores[ listaColores.getSelectedIndex() ] );
47              }
48
49           } // fin de la clase interna anónima
50
51        ); // fin de la llamada a addListSelectionListener
52
53        setSize( 350, 150 );
54        setVisible( true );
55
56     } // fin del constructor de PruebaLista
57
58     public static void main( String args[] )
59     {
60        PruebaLista aplicacion = new PruebaLista();
61        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
62     }
63
64  } // fin de la clase PruebaLista
```

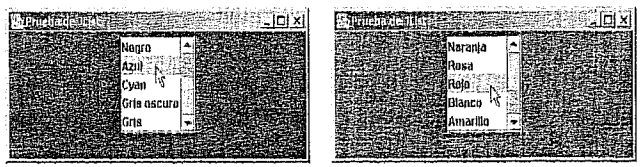


Figura 13.14 Objeto `JList` que muestra una lista de colores. (Parte 2 de 2.)

Un objeto `JList` se crea en la línea 30 y se asigna a la referencia `listaColores` mediante el constructor. El argumento para el constructor de `JList` es el arreglo de objetos `Object` (en este caso cadenas) a mostrar en la lista. En la línea 31 se usa el método ***`setVisibleRowCount`*** de `JList` para determinar el número de elementos que serán visibles en la lista.

En la línea 34 se utiliza el método ***`setSelectionMode`*** de `JList` para especificar el *modo de selección* de la lista. La clase ***`ListSelectionModel`*** (del paquete `javax.swing`) declara tres constantes que especifican el modo de selección de un objeto `JList`: ***`SINGLE_SELECTION`*** (que permite seleccionar solamente un elemento en un momento dado), ***`SINGLE_INTERVAL_SELECTION`*** (para una lista de selección

múltiple que permita la selección de varios elementos contiguos) y ***`MULTIPLE_INTERVAL_SELECTION`*** (para una lista de selección múltiple que no restrinja el número de elementos que pueden seleccionarse).

A diferencia de un objeto `JComboBox`, un objeto `JList` *no* proporciona una barra de desplazamiento si hay más elementos en la lista que el número de filas visibles. En este caso, se utiliza un objeto ***`JScrollPane`*** para proporcionar la capacidad de desplazamiento al objeto `JList`. En la línea 37 se agrega una nueva instancia de la clase ***`JScrollPane`*** al panel de contenido. El constructor de `JScrollPane` recibe como su argumento el objeto `JComponent` que necesita la funcionalidad de desplazamiento (en este caso, `listaColores`). Observe en las capturas de pantalla que aparece una barra de desplazamiento, creada por el objeto `JScrollPane`, del lado derecho del objeto `JList`. De manera predeterminada, la barra de desplazamiento aparece solamente cuando el número de elementos en el objeto `JList` es mayor que el número de elementos visibles.

En las líneas 38 a 51 se utiliza el método ***`addListSelectionListener`*** de `JList` para registrar una instancia de una clase interna anónima que implementa a ***`ListSelectionListener`*** (en el paquete `javax.swing.event`) como el componente de escucha para `listaColores`. Cuando el usuario selecciona un elemento de la `listaColores`, el método ***`valueChanged`*** (líneas 43 a 47) se ejecuta y establece el color de fondo del panel de contenido, mediante el método ***`setBackground`***. Recuerde que el panel de contenido es un objeto `Container`. La clase `Container` hereda el método `setBackground` de la clase `Component`. El color se selecciona del arreglo `colores`, utilizando el índice del elemento seleccionado. El método `getSelectedIndex` de `JList` devuelve el índice del elemento seleccionado. Al igual que con los arreglos y los objetos `JComboBox`, los índices en `JList` están basados en cero.

13.11 Listas de selección múltiple

Una *lista de selección múltiple* permite al usuario seleccionar varios elementos de un objeto `JList`. Una lista `SINGLE_INTERVAL_SELECTION` permite la selección de un rango contiguo de elementos en la lista. Para ello, haga clic en el primer elemento y después oprima (y mantenga oprimida) la tecla *Mayús* mientras hace clic en el último elemento a seleccionar en el rango. Una lista `MULTIPLE_INTERVAL_SELECTION` permite una selección de rango continuo, como se describe para una lista `SINGLE_INTERVAL_SELECTION`. Dicha lista permite que se seleccionen diversos elementos, oprimiendo y manteniendo oprimida la tecla *Ctrl* (a la que algunas veces se le conoce como la tecla de *Control*) mientras hace clic en cada elemento a seleccionar. Para deseleccionar un elemento, oprima y mantenga oprimida la tecla *Ctrl* mientras hace clic en el elemento por segunda vez.

La aplicación de la figura 13.15 utiliza listas de selección múltiple para copiar elementos de un objeto `JList` a otro. Una lista es de tipo `MULTIPLE_INTERVAL_SELECTION` y la otra es de tipo `SINGLE_INTERVAL_SELECTION`. Cuando ejecute el programa, trate de usar las técnicas de selección descritas anteriormente para seleccionar elementos en ambas listas.

```
1  // Fig. 13.15: PruebaSeleccionMultiple.java
2  // Cómo copiar elementos de una lista a otra.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaSeleccionMultiple extends JFrame {
8     private JList listaColores, listaCopia;
9     private JButton botonCopiar;
10    private final String nombresColores[] = { "Negro", "Azul", "Cyan",
11       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta", "Naranja",
12       "Rosa", "Rojo", "Blanco", "Amarillo" };
13
14    // configurar la GUI
```

Figura 13.15 Objeto `JList` que permite selecciones múltiples. (Parte 1 de 3.)

http://librosysolucionarios.net

```
15    public PruebaSeleccionMultiple()
16    {
17       super( "Listas de selección múltiple" );
18
19       // obtener panel de contenido y establecer su esquema
20       Container contenedor = getContentPane();
21       contenedor.setLayout( new FlowLayout() );
22
23       // establecer objeto JList listaColores
24       listaColores = new JList( nombresColores );
25       listaColores.setVisibleRowCount( 5 );
26       listaColores.setSelectionMode(
27          ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
28       contenedor.add( new JScrollPane( listaColores ) );
29
30       // crear botón copiar y registrar su componente de escucha
31       botonCopiar = new JButton( "Copiar >>>" );
32       botonCopiar.addActionListener(
33
34          new ActionListener() {  // clase interna anónima
35
36             // manejar evento de botón
37             public void actionPerformed( ActionEvent evento )
38             {
39                // colocar valores seleccionados en listaCopia
40                listaCopia.setListData( listaColores.getSelectedValues() );
41             }
42
43          } // fin de clase interna anónima
44
45       ); // fin de la llamada a addActionListener
46
47       contenedor.add( botonCopiar );
48
49       // establecer objeto JList listaCopia
50       listaCopia = new JList( );
51       listaCopia.setVisibleRowCount( 5 );
52       listaCopia.setFixedCellWidth( 100 );
53       listaCopia.setFixedCellHeight( 15 );
54       listaCopia.setSelectionMode(
55          ListSelectionModel.SINGLE_INTERVAL_SELECTION );
56       contenedor.add( new JScrollPane( listaCopia ) );
57
58       setSize( 325, 130 );
59       setVisible( true );
60
61    } // fin del constructor PruebaSeleccionMultiple
62
63    public static void main( String args[] )
64    {
65       PruebaSeleccionMultiple aplicacion = new PruebaSeleccionMultiple();
66       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
67    }
68
69 } // fin de la clase PruebaSeleccionMultiple
```

Figura 13.15 Objeto `JList` que permite selecciones múltiples. (Parte 2 de 3.)

http://librosysolucionarios.net

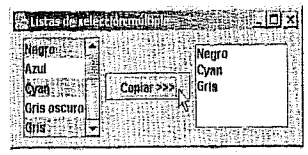


Figura 13.15 Objeto `JList` que permite selecciones múltiples. (Parte 3 de 3.)

En la línea 24 se crea el objeto `JList` llamado **`listaColores`** y se inicializa con las cadenas en el arreglo **`nombresColores`**. En la línea 25 se establece el número de filas visibles en **`listaColores`** a **5**. En las líneas 26 y 27 se especifica que **`listaColores`** es una lista de tipo **`MULTIPLE_INTERVAL_SELECTION`**. En la línea 28 se agrega un nuevo objeto **`JScrollPane`**, que contiene **`listaColores`**, al panel de contenido. En las líneas 50 a 56 se realizan tareas similares para **`listaCopia`**, la cual se declara como una lista tipo **`SINGLE_INTERVAL_SELECTION`**. En la línea 52 se utiliza el método ***`setFixedCellWidth`*** de **`JList`** para establecer la anchura de **`listaCopia`** en 100 píxeles. En la línea 53 se utiliza el método ***`setFixedCellHeight`*** de **`JList`** para establecer la altura de cada elemento en el objeto **`JList`** a 15 píxeles.

Una lista de selección múltiple no tiene un evento específico asociado para hacer selecciones múltiples. Normalmente, un evento generado por otro componente de la GUI (lo que se conoce como un *evento externo*) especifica cuándo deben procesarse las selecciones múltiples en un objeto **`JList`**. En este ejemplo, el usuario hace clic en el objeto **`JButton`** llamado **`botonCopiar`** para desencadenar el evento que copia los elementos seleccionados en **`listaColores`** a **`listaCopia`**.

Cuando el usuario hace clic en **`botonCopiar`**, el método **`actionPerformed`** (líneas 37 a 41) utiliza el método ***`setListData`*** de **`JList`** para establecer los elementos mostrados en **`listaCopia`**. En la línea 40 se hace una llamada al método ***`getSelectedValues`*** de **`listaColores`**, el cual devuelve un arreglo de objetos **`Object`** que representan los elementos seleccionados en **`listaColores`**. En este ejemplo, el arreglo devuelto se pasa como argumento al método **`setListData`** de **`listaCopia`**.

Muchos estudiantes preguntan cómo puede usarse la referencia **`listaCopia`** en la línea 40, aún y cuando el programa no crea el objeto al que hace referencia, sino hasta la línea 50. Recuerde que el método **`actionPerformed`** (líneas 37 a 41) no se ejecuta sino hasta que el usuario oprime el **`botonCopia`**, lo cual no puede ocurrir sino hasta que el constructor termina de ejecutarse y el programa muestra la GUI. En ese punto de la ejecución del programa, en la línea 40 ya se ha inicializado **`listaCopia`** con un nuevo objeto **`JList`**. Ningún método se ejecuta sino hasta que es llamado. En el caso de un manejador de eventos, la llamada al método ocurre en respuesta a un evento.

13.12 Manejo de eventos de ratón

En esta sección presentaremos las interfaces de escucha de eventos ***`MouseListener`*** y ***`MouseMotionListener`*** para manejar *eventos de ratón*. Estos eventos pueden atraparse para cualquier componente de la GUI que se derive de **`java.awt.Component`**. Los métodos de las interfaces **`MouseListener`** y **`MouseMotionListener`** se sintetizan en la figura 13.16.[1]

Métodos de las interfaces `MouseListener` y `MouseMotionListener`

Métodos de la interfaz `MouseListener`

`public void mousePressed( MouseEvent evento )`

Se llama cuando se oprime un botón del ratón, mientras el cursor del ratón está sobre un componente.

Figura 13.16 Métodos de la interfaz **`MouseListener`** y **`MouseMotionListener`**. (Parte 1 de 2.)

1. A partir de Java 1.4, el paquete **`javax.swing.event`** contiene la interfaz **`MouseInputListener`**, que extiende a las interfaces **`MouseListener`** y **`MouseMotionListener`** para crear una sola interfaz que contiene todos los métodos de **`MouseListener`** y **`MouseMotionListener`**.

Métodos de las interfaces `MouseListener` y `MouseMotionListener`

`public void mouseClicked( MouseEvent evento )`

Se llama cuando se oprime y suelta un botón del ratón, mientras el cursor del ratón permanece estacionario sobre un componente.

`public void mouseReleased( MouseEvent evento )`

Se llama cuando se suelta un botón de ratón después de ser oprimido. Este evento siempre sigue después de un evento `mousePressed`.

`public void mouseEntered( MouseEvent evento )`

Se llama cuando el cursor del ratón entra a los límites de un componente.

`public void mouseExited( MouseEvent evento )`

Se llama cuando el cursor del ratón sale de los límites de un componente.

Métodos de la interfaz `MouseMotionListener`

`public void mouseDragged( MouseEvent evento )`

Se llama cuando el botón del ratón se oprime mientras el cursor del ratón se encuentra sobre un componente y se mueve mientras el botón sigue oprimido. Este evento siempre sigue después de una llamada a `mousePressed`. Todos los eventos de arrastre del ratón se envían al componente en el cual empezó la acción de arrastre.

`public void mouseMoved( MouseEvent evento )`

Se llama al moverse el ratón cuando su cursor se encuentra sobre un componente. Todos los eventos de movimiento se envían al componente sobre el cual se encuentra el ratón posicionado en ese momento.

Figura 13.16 Métodos de la interfaz `MouseListener` y `MouseMotionListener`. (Parte 2 de 2.)

Cada uno de los métodos manejadores de eventos de ratón toma un objeto ***`MouseEvent`*** como su argumento. Un objeto `MouseEvent` contiene información acerca del evento de ratón que ocurrió, incluyendo las coordenadas *x* y *y* de la ubicación en donde ocurrió el evento. Los métodos de `MouseListener` y `MouseMotionListener` son llamados cuando el ratón interactúa con un objeto `Component`, si hay registrados objetos de escucha apropiados para ese objeto `Component`. El método ***`mousePressed`*** se llama cuando se oprime un botón del ratón, mientras el cursor está sobre un componente. Al utilizar los métodos y constantes de la clase ***`InputEvent`*** (la superclase de `MouseEvent`), un programa puede determinar cuál fue el botón del ratón que oprimió el usuario.

Observación de apariencia visual 13.8

Las llamadas al método `mouseDragged` se envían al objeto `MouseMotionListener` para el objeto `Component` en el que empezó la operación de arrastre. De manera similar, la llamada al método `mouseReleased` al final de una operación de arrastre se envía al objeto `MouseListener` para el objeto `Component` en el que empezó la operación de arrastre.

Además de las interfaces `MouseListener` y `MouseMotionListener`, Java ahora cuenta con la interfaz ***`MouseWheelListener`*** para permitir a los programas responder a la rotación del disco en un ratón que tenga uno. Esta interfaz declara el método ***`mouseWheelMoved`***, el cual recibe un evento ***`MouseWheelEvent`*** como argumento. La clase `MouseWheelEvent` (una subclase de `MouseEvent`) contiene métodos que permiten al manejador de eventos obtener información acerca de la cantidad de rotación del disco.

La aplicación `RastreadorRaton` (figura 13.17) demuestra el uso de los métodos de las interfaces `MouseListener` y `MouseMotionListener`. La clase de aplicación implementa ambas interfaces, para poder escuchar sus propios eventos de ratón. Observe que los siete métodos de estas interfaces deben ser declarados por el programador cuando una clase implementa ambas interfaces. Cada evento de ratón en este ejemplo muestra una cadena en el objeto `JLabel` llamado `barraEstado`, en la parte inferior de la ventana.


```
1  // Fig. 13.17: RastreadorRaton.java
2  // Demostración de los eventos de ratón.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class RastreadorRaton extends JFrame
8     implements MouseListener, MouseMotionListener {
9
10    private JLabel barraEstado;
11
12    // configurar la GUI y registrar manejadores de eventos de ratón
13    public RastreadorRaton()
14    {
15       super( "Demostración de los eventos de ratón" );
16
17       barraEstado = new JLabel();
18       getContentPane().add( barraEstado, BorderLayout.SOUTH );
19
20       addMouseListener( this );       // escucha sus propios eventos de ratón
21       addMouseMotionListener( this ); // y de movimiento de ratón
22
23       setSize( 300, 125 );
24       setVisible( true );
25    }
26
27    // Manejadores de eventos de MouseListener
28    // manejar el evento cuando el botón del ratón se suelta inmediatamente después
         de oprimir
29    public void mouseClicked( MouseEvent evento )
30    {
31       barraEstado.setText( "Se hizo clic en [" + evento.getX() +
32          ", " + evento.getY() + "]" );
33    }
34
35    // manejar evento cuando se oprime el botón del ratón
36    public void mousePressed( MouseEvent evento )
37    {
38       barraEstado.setText( "Se oprimió en [" + evento.getX() +
39          ", " + evento.getY() + "]" );
40    }
41
42    // manejar evento cuando se suelta el ratón después de arrastrar
43    public void mouseReleased( MouseEvent evento )
44    {
45       barraEstado.setText( "Se soltó en [" + evento.getX() +
46          ", " + evento.getY() + "]" );
47    }
48
49    // manejar el evento cuando el ratón entra al área
50    public void mouseEntered( MouseEvent evento )
51    {
52       barraEstado.setText( "Ratón entro en [" + evento.getX() +
53          ", " + evento.getY() + "]" );
54       getContentPane().setBackground( Color.GREEN );
55    }
```

Figura 13.17 Manejo de eventos de ratón. (Parte 1 de 2.)

http://librosysolucionarios.net

```
56
57    // manejar evento cuando el ratón sale del área
58    public void mouseExited( MouseEvent evento )
59    {
60       barraEstado.setText( "Ratón fuera de la ventana" );
61       getContentPane().setBackground( Color.WHITE );
62    }
63
64    // Manejadores de eventos de MouseMotionListener
65    // manejar el evento cuando el usuario arrastra el ratón con el botón oprimido
66    public void mouseDragged( MouseEvent evento )
67    {
68       barraEstado.setText( "Se arrastró en [" + evento.getX() +
69          ", " + evento.getY() + "]" );
70    }
71
72    // manejar el evento cuando el usuario mueve el ratón
73    public void mouseMoved( MouseEvent evento )
74    {
75       barraEstado.setText( "Se movió en [" + evento.getX() +
76          ", " + evento.getY() + "]" );
77    }
78
79    public static void main( String args[] )
80    {
81       RastreadorRaton aplicacion = new RastreadorRaton();
82       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
83    }
84
85 } // fin de la clase RastreadorRaton
```

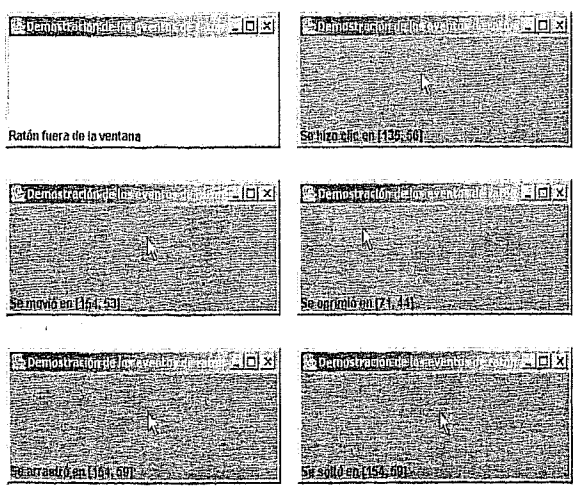


Figura 13.17 Manejo de eventos de ratón. (Parte 2 de 2.)

En las líneas 17 y 18, en el constructor, se declara el objeto `JLabel` llamado `barraEstado` y se adjunta al panel de contenido. Hasta ahora, cada vez que utilizamos el panel de contenido se hizo una llamada a `setLayout` para establecer el administrador de esquemas del panel de contenido en `FlowLayout`. Esto permite al panel de contenido mostrar los componentes de la GUI que le adjuntamos, de izquierda a derecha. Si los componentes de la GUI no caben en una línea, el esquema `FlowLayout` permite que éstos fluyan hacia líneas adicionales. En realidad, el administrador de esquemas predeterminado es ***`BorderLayout`***, el cual divide el área del panel de contenido en cinco regiones: norte, sur, este, oeste y centro. En la línea 18 se utiliza una nueva versión del método `add` de `Container` para agregar `barraEstado` a la región ***`BorderLayout.SOUTH`***, que se extiende a lo largo de la parte inferior del panel de contenido. Hablaremos sobre `BorderLayout` y otros administradores de esquemas detalladamente en la sección 13.15.

En las líneas 20 y 21 del constructor se registra el objeto `RastreadorRaton` como el componente de escucha para sus propios eventos de ratón. Los métodos ***`addMouseListener`*** y ***`addMouseMotionListener`*** son métodos de `Component` que pueden utilizarse para registrar componentes de escucha de eventos de ratón para un objeto de cualquier clase que extienda a `Component` (como lo hace `RastradorRaton` en forma indirecta).

Cuando el ratón entra o sale del área de la aplicación, se hacen llamadas a los métodos `mouseEntered` (líneas 50 a 55) y `mouseExited` (líneas 58 a 62), respectivamente. El método `mouseEntered` muestra un mensaje en `barraEstado` indicando que el ratón entró a la ventana, y cambia el color de fondo a verde. El método `mouseExited` muestra un mensaje en `barraEstado` indicando que el ratón está fuera de la aplicación (vea la primera ventana de resultados de ejemplo) y cambia el color de fondo a blanco.

Cuando ocurre alguno de los otros cinco eventos, se muestra un mensaje en `barraEstado` que incluye una cadena, la cual contiene el evento que ocurrió y las coordenadas en donde ocurrió este evento de ratón. Las coordenadas *x* y *y* del ratón cuando ocurrió el evento se obtienen con los métodos ***`getX`*** y ***`getY`*** de `MouseEvent`, respectivamente.

13.13 Clases adaptadoras

Muchas de las interfaces de escucha de eventos proporcionan varios métodos; `MouseListener` y `MouseMotionListener` son dos ejemplos. No siempre es deseable declarar todos los métodos en una interfaz de escucha de eventos. Por ejemplo, un programa podría necesitar solamente el manejador `mouseClicked` de la interfaz `MouseListener`, o el manejador `mouseDragged` de la interfaz `MouseMotionListener`. En la figura 10.23 demostramos que `windowClosing` es llamado cuando el usuario cierra una ventana. La interfaz `WindowListener`, que contiene el método `windowClosing`, especifica siete métodos manejadores de eventos. Para muchas de las interfaces de escucha de eventos que contienen varios métodos, el paquete `java.awt.event` y el paquete `javax.swing.event` proporcionan *clases adaptadoras* de escucha de eventos. Una clase adaptadora implementa a una interfaz y proporciona una implementación predeterminada (con un cuerpo vacío para los métodos) de todos los métodos en la interfaz. En la figura 13.18 se muestran varias clases adaptadoras de `java.awt.event`, junto con las interfaces que implementan.

El programador puede extender la clase adaptadora para heredar la implementación predeterminada de cada método y, por ende, sobrescribir el (los) método(s) necesario(s) para el manejo de eventos. La implementación predeterminada de cada método en la clase adaptadora tiene un cuerpo vacío.

Clase adaptadora de eventos	Implementa a la interfaz
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Figura 13.18 Clases adaptadoras de eventos y las interfaces que implementan en el paquete `java.awt.event`.

http://librosysolucionarios.net

Observación de ingeniería de software 13.3

Cuando una clase implementa a una interfaz, la clase tiene una relación del tipo "es un" con esa interfaz. Todas las subclases directas e indirectas de esa clase heredan esta interfaz. Por lo tanto, un objeto de una clase que extiende a una clase adaptadora de eventos es un objeto del tipo de escucha de eventos correspondiente (por ejemplo, un objeto de una subclase de `MouseAdapter` es un `MouseListener`).

Extensión de `MouseMotionAdapter`

La aplicación `Pintor` de la figura 13.19 utiliza el manejador de eventos `mouseDragged` para crear un programa simple de dibujo. El usuario puede dibujar imágenes con el ratón, arrastrándolo en el fondo de la ventana. Este ejemplo no utiliza el método `mouseMoved`, por lo que nuestro objeto `MouseMotionListener` se declara como una subclase de `MouseMotionAdapter`. Esta clase ya declara tanto a `mouseMoved` como a `mouseDragged`, por lo que simplemente podemos sobrescribir a `mouseDragged` para proporcionar la funcionalidad requerida para dibujar.

```
1  // Fig. 13.19: Pintor.java
2  // Uso de la clase MouseMotionAdapter.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class Pintor extends JFrame {
8     private int cuentaPuntos = 0;
9
10    // arreglo de 1000 referencias a java.awt.Point
11    private Point puntos[] = new Point[ 1000 ];
12
13    // configurar la GUI y registrar manejador de eventos de ratón
14    public Pintor()
15    {
16       super( "Un programa simple de dibujo" );
17
18       // crear una etiqueta y colocarla en la parte SOUTH del esquema BorderLayout
19       getContentPane().add( new JLabel( "Arrastre el ratón para dibujar" ),
20          BorderLayout.SOUTH );
21
22       addMouseMotionListener(
23
24          new MouseMotionAdapter() {  // clase interna anónima
25
26             // almacenar coordenadas de arrastre y llamar a repaint
27             public void mouseDragged( MouseEvent evento )
28             {
29                if ( cuentaPuntos < puntos.length ) {
30                   puntos[ cuentaPuntos ] = evento.getPoint();
31                   ++cuentaPuntos;
32                   repaint();
33                }
34             }
35
36          } // fin de la clase interna anónima
37
38       ); // fin de la llamada a addMouseMotionListener
39
40       setSize( 300, 150 );
41       setVisible( true );
```

Figura 13.19 Clases adaptadoras utilizadas para implementar manejadores de eventos. (Parte 1 de 2.)

http://librosysolucionarios.net


```
42
43    } // fin del constructor de Pintor
44
45    // dibujar óvalo en un cuadro delimitador de 4 por 4 en ubicación especificada en ventana
46    public void paint( Graphics g )
47    {
48       super.paint( g ); // borra el área de dibujo
49
50       for ( int i = 0; i < puntos.length && puntos[ i ] != null; i++ )
51          g.fillOval( puntos[ i ].x, puntos[ i ].y, 4, 4 );
52    }
53
54    public static void main( String args[] )
55    {
56       Pintor aplicacion = new Pintor();
57       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
58    }
59
60 } // fin de la clase Pintor
```

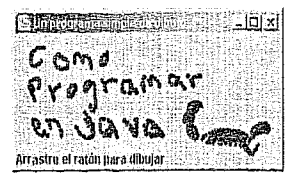


Figura 13.19 Clases adaptadoras utilizadas para implementar manejadores de eventos. (Parte 2 de 2.)

En este ejemplo utilizamos un arreglo de 1000 objetos `java.awt.Point` para almacenar la ubicación en la que ocurre cada evento de arrastre del ratón. Esto permite al método `paint` dibujar un pequeño óvalo en cada ubicación almacenada en el arreglo. En las líneas 22 a 38 se registra un objeto `MouseMotionListener` para escuchar los eventos de movimiento del ratón de la ventana. Recuerde que, cuando se invoca un método utilizando sólo su nombre, en realidad se invoca mediante la referencia `this`, indicando que ese método es llamado para la instancia actual de la clase en tiempo de ejecución. En las líneas 24 a 36 se declara una clase interna anónima que extiende a la clase `MouseMotionAdapter` (la cual implementa a `MouseMotionListener`). La clase interna anónima hereda una implementación predeterminada de los métodos `mouseMoved` y `mouseDragged`. Por lo tanto, la clase interna anónima ya satisface el requerimiento que establece que, en todos los métodos, debe implementarse una interfaz. Sin embargo, los métodos predeterminados no hacen nada cuando son llamados. Así que debemos sobrescribir el método `mouseDragged` en las líneas 27 a 34 para capturar las coordenadas de un evento de arrastre del ratón y almacenarlas como un objeto de la clase `Punto`. En la línea 29 se asegura que almacenemos las coordenadas del evento solamente si hay elementos vacíos en el arreglo. De ser así, en la línea 30 se invoca el método `getPoint` del objeto `MouseEvent` para obtener el objeto `Point` en donde ocurrió el evento y lo guarda en el arreglo, en el índice `cuentaPuntos`. En la línea 31 se incrementa la `cuentaPuntos` y en la 32 se hace una llamada a `repaint` para empezar a dibujar. Observe que `paint` (líneas 46 a 52) itera a través de todos los objetos `Point` en el arreglo y dibuja un óvalo en cada ubicación. La clase `Point` tiene dos variables de instancia públicas (`x` y `y`) que se utilizan directamente en la línea 51 para especificar la esquina superior izquierda del cuadro delimitador del óvalo. El ciclo termina, ya sea cuando se encuentra una referencia `null` en el arreglo o cuando se llega al final del mismo.

Extensión de `MouseAdapter`

La aplicación `DetallesRaton` de la figura 13.20 demuestra cómo determinar el número de clics del ratón (es decir, la cuenta de clics) y cómo diferenciar entre los distintos botones del ratón. El componente de escu-

cha de eventos en este programa es un objeto de la clase interna **ManejadorClicsRaton** (líneas 38 a 62), el cual extiende a **MouseAdapter** para que podamos declarar sólo el método **mouseClicked** que necesitamos en este ejemplo.

```
1  // Fig. 13.20: DetallesRaton.java
2  // Demostración de los clics de ratón y cómo diferenciar entre los botones del ratón.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DetallesRaton extends JFrame {
8     private int xPos, yPos;
9
10    // establecer cadena barra título; registrar escucha ratón; ajustar tamaño y
        mostrar ventana
11    public DetallesRaton()
12    {
13       super( "Clics y botones del ratón" );
14
15       addMouseListener( new ManejadorClicsRaton() );
16
17       setSize( 350, 150 );
18       setVisible( true );
19    }
20
21    // dibujar objeto String en la ubicación donde se hizo clic con el ratón
22    public void paint( Graphics g )
23    {
24       // llamar al método paint de la superclase
25       super.paint( g );
26
27       g.drawString( "Se hizo clic en: [" + xPos + ", " + yPos + "]",
28          xPos, yPos );
29    }
30
31    public static void main( String args[] )
32    {
33       DetallesRaton aplicacion = new DetallesRaton();
34       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
35    }
36
37    // clase interna para manejar eventos de ratón
38    private class ManejadorClicsRaton extends MouseAdapter {
39
40       // manejar evento de clic del ratón y determinar cuál botón se oprimió
41       public void mouseClicked( MouseEvent evento )
42       {
43          xPos = evento.getX();
44          yPos = evento.getY();
45
46          String titulo = "Se hizo clic " + evento.getClickCount() + " Veces";
47
48          if ( evento.isMetaDown() )  // botón derecho del ratón
49             titulo += " con el botón derecho del ratón";
50
51          else if ( evento.isAltDown() )  // botón de en medio del ratón
```

Figura 13.20 Clics con los botones izquierdo, central y derecho del ratón. (Parte 1 de 2.)


```
52             titulo += " con el botón central del ratón";
53
54          else  // botón izquierdo del ratón
55             titulo += " con el botón izquierdo del ratón";
56
57          setTitle( titulo );  // establecer barra de título de la ventana
58          repaint();
59
60       } // fin del método mouseClicked
61
62    } // fin de la clase interna privada ManejadorClicsRaton
63
64 } // fin de la clase DetallesRaton
```

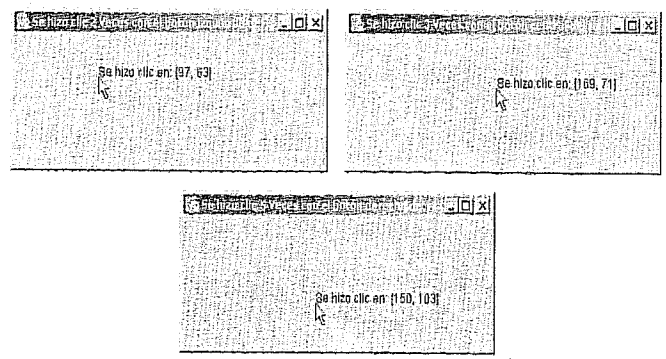


Figura 13.20 Clics con los botones izquierdo, central y derecho del ratón. (Parte 2 de 2.)

Un usuario de un programa de Java podría estar en un sistema con un ratón de uno, dos o tres botones. Java proporciona un mecanismo para diferenciar entre los botones del ratón. La clase **MouseEvent** hereda varios métodos de la clase **InputEvent**, los cuales pueden diferenciar entre los botones del ratón en un ratón con varios botones, o pueden imitar un ratón con varios ratones mediante una combinación de pulsación de tecla con un clic de botón del ratón. En la figura 13.21 se muestran los métodos de **InputEvent** que se utilizan para diferenciar entre los clics de los botones del ratón. Java supone que todo ratón contiene un botón izquierdo. Por lo tanto, es sencillo probar un clic de botón izquierdo del ratón. Sin embargo, los usuarios con un ratón de uno o dos botones deben usar una combinación de pulsaciones de tecla y clics de botón del ratón al

Método de `InputEvent`	Descripción
`isMetaDown()`	Devuelve **true** cuando el usuario hace clic con el botón derecho del ratón, en un ratón con dos o tres botones. Para simular un clic con el botón derecho en un ratón con un solo botón, el usuario puede mantener oprimida la tecla *Meta* en el teclado y hacer clic en el botón del ratón.
`isAltDown()`	Devuelve **true** cuando el usuario hace clic con el botón de en medio del ratón, en un ratón con tres botones. Para simular un clic con el botón de en medio en un ratón con uno o dos botones, el usuario puede oprimir la tecla *Alt* en el teclado y hacer clic con el único botón, o el botón izquierdo del ratón, respectivamente.

Figura 13.21 Métodos de **InputEvent** que ayudan a diferenciar entre los clics de los botones izquierdo, central y derecho del ratón.

http://librosysolucionarios.net

mismo tiempo para simular los botones que faltan en el ratón. En el caso de un ratón de uno o dos botones, este programa supone que se hace clic en el botón central del ratón si el usuario mantiene oprimida la tecla *Alt* y hace clic con el botón izquierdo del ratón en un ratón con dos botones, o en el único botón en un ratón con un solo botón. En el caso de un ratón con un botón, este programa supone que se hace clic con el botón derecho del ratón si el usuario mantiene oprimida la tecla *Meta* y hace clic en el botón del ratón.

El método **mouseClicked** (líneas 41 a 60 de la figura 13.20) captura primero las coordenadas en donde ocurrió el evento y las almacena en las variables de instancia **xPos** y **yPos** de la clase **DetallesRaton**. En la línea 46 se crea una cadena que contiene el número de clics del ratón (según el valor que devuelve el método ***getClickCount*** de **MouseEvent** en la línea 46). La instrucción **if** anidada en las líneas 48 a 55 utiliza los métodos **isMetaDown** e **isAltDown** para determinar con cuál botón hizo clic el usuario, y adjunta una cadena apropiada a **titulo** en cada caso. La cadena resultante se muestra en la barra de título de la ventana, utilizando el método ***setTitle*** (que se hereda en la clase **JFrame** de la clase **Frame**) en la línea 57. En la línea 58 se hace una llamada a **repaint** para que inicie, a su vez, una llamada a **paint** para dibujar una cadena en la ubicación en donde el usuario hizo clic con el botón del ratón.

13.14 Manejo de eventos de teclas

En esta sección presentamos la interfaz **KeyListener** para manejar *eventos de teclas*. Estos eventos se generan cuando se oprimen y sueltan las teclas en el teclado. Una clase que implementa a **KeyListener** debe proporcionar declaraciones para los métodos ***keyPressed***, ***keyReleased*** y ***keyTyped***, cada uno de los cuales recibe un objeto ***KeyEvent*** como argumento. La clase **KeyEvent** es una subclase de **InputEvent**. El método **keyPressed** es llamado en respuesta a la acción de oprimir cualquier tecla. El método **keyTyped** es llamado en respuesta a la acción de oprimir una tecla que no sea una *tecla de acción*. (Las teclas de acción son cualquier tecla de dirección, *Inicio*, *Fin*, *Re Pág*, *Av Pág*, cualquier tecla de función, *Bloq Num*, *Impr Pant*, *Bloq Despl*, *Bloq Mayús* y *Pausa*.) El método **keyReleased** es llamado cuando la tecla se suelta después de un evento **keyPressed** o **keyTyped**.

En la figura 13.22 se demuestra el uso de los métodos de **KeyListener**. La clase **DemoTeclas** implementa la interfaz **KeyListener**, por lo que los tres métodos se declaran en la aplicación.

```
1  // Fig. 13.22: DemoTeclas.java
2  // Demostración de los eventos de pulsación de tecla.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoTeclas extends JFrame implements KeyListener {
8     private String linea1 = "", linea2 = "", linea3 = "";
9     private JTextArea areaTexto;
10
11    // configurar la GUI
12    public DemoTeclas()
13    {
14       super( "Demostración de eventos de pulsación de tecla" );
15
16       // establecer objeto JTextArea
17       areaTexto = new JTextArea( 10, 15 );
18       areaTexto.setText( "Oprima cualquier tecla en el teclado..." );
19       areaTexto.setEnabled( false );
20       areaTexto.setDisabledTextColor( Color.BLACK );
21       getContentPane().add( areaTexto );
22
23       addKeyListener( this ); // permitir al marco procesar eventos de teclas
```

Figura 13.22 Manejo de eventos de teclas. (Parte 1 de 3.)


```
24
25       setSize( 350, 100 );
26       setVisible( true );
27
28    } // fin del constructor de DemoTeclas
29
30    // manejar evento de pulsación de cualquier tecla
31    public void keyPressed( KeyEvent evento )
32    {
33       linea1 = "Se oprimió tecla: " + evento.getKeyText( evento.getKeyCode() );
34       establecerLineas2y3( evento );
35    }
36
37    // manejar evento de liberación de cualquier tecla
38    public void keyReleased( KeyEvent evento )
39    {
40       linea1 = "Se soltó tecla: " + evento.getKeyText( evento.getKeyCode() );
41       establecerLineas2y3( evento );
42    }
43
44    // manejar evento de pulsación de una tecla de acción
45    public void keyTyped( KeyEvent evento )
46    {
47       linea1 = "Se escribió tecla: " + evento.getKeyChar();
48       establecerLineas2y3( evento );
49    }
50
51    // establecer segunda y tercera líneas de salida
52    private void establecerLineas2y3( KeyEvent evento )
53    {
54       linea2 = "Esta tecla " + ( evento.isActionKey() ? "" : "no " ) +
55          "es una tecla de acción";
56
57       String temp = evento.getKeyModifiersText( evento.getModifiers() );
58
59       linea3 = "Teclas modificadoras oprimidas: " +
60          ( temp.equals( "" ) ? "ninguna" : temp );
61
62       areaTexto.setText( linea1 + "\n" + linea2 + "\n" + linea3 + "\n" );
63    }
64
65    public static void main( String args[] )
66    {
67       DemoTeclas aplicacion = new DemoTeclas();
68       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
69    }
70
71 } // fin de la clase DemoTeclas
```

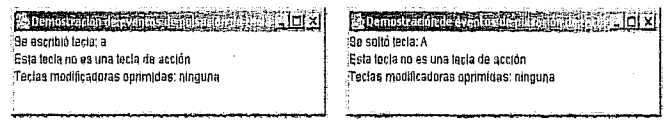


Figura 13.22 Manejo de eventos de teclas. (Parte 2 de 3.)

http://librosysolucionarios.net

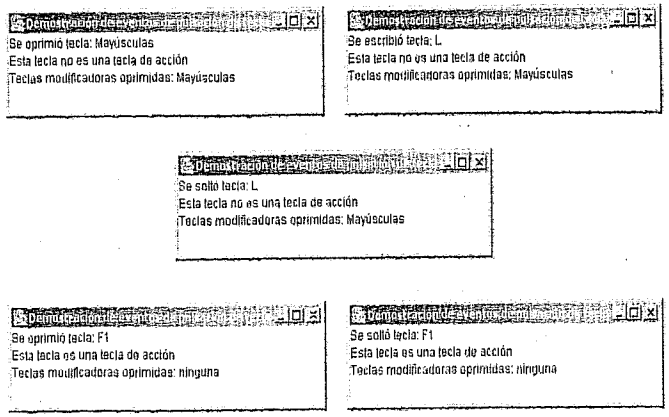


Figura 13.22 Manejo de eventos de teclas. (Parte 3 de 3.)

El constructor (líneas 12 a 28) registra a la aplicación para manejar sus propios eventos de teclas, utilizando el método `addKeyListener` en la línea 23. Este método se declara en la clase `Component`, por lo que todas las subclases de `Component` pueden notificar a objetos `KeyListener` acerca de los eventos para ese objeto `Component`.

En la línea 21, en el constructor, se agrega el objeto `JTextArea` llamado `areaTexto` (en donde se muestra la salida del programa) al panel de contenido. Observe en las capturas de pantalla que `areaTexto` ocupa toda la ventana. Esto se debe al esquema predeterminado `BorderLayout` del panel de contenido (que describiremos en la sección 13.15.2 y demostraremos en la figura 13.25). Cuando se agrega un objeto `Component` individual a un objeto `BorderLayout`, el objeto `Component` ocupa todo el objeto `Container` completo. Observe que en la línea 20 se utiliza el método `setDisabledTextColor` para cambiar el color del texto en el área de texto a negro.

Los métodos `keyPressed` (líneas 31 a 35) y `keyReleased` (líneas 38 a 42) utilizan el método `getKeyCode` de `KeyEvent` para obtener el *código de tecla virtual* de la tecla oprimida. La clase `KeyEvent` mantiene un conjunto de constantes (las constantes de código de tecla virtual) que representan a todas las teclas en el teclado. Estas constantes pueden compararse con el valor de retorno de `getKeyCode` para probar teclas individuales en el teclado. El valor devuelto por `getKeyCode` se pasa al método `getKeyText` de `KeyEvent`, el cual devuelve una cadena que contiene el nombre de la tecla que se oprimió. Para obtener una lista completa de las constantes de teclas virtuales, vea la documentación en línea para la clase `KeyEvent` (paquete `java.awt.event`). El método `keyTyped` (líneas 45 a 49) utiliza el método `getKeyChar` de `KeyEvent` para obtener el valor Unicode del carácter escrito.

Los tres métodos manejadores de eventos terminan llamando al método `establecerLineas2y3` (líneas 52 a 63) y le pasan el objeto `KeyEvent`. Este método utiliza el método `isActionKey` de `KeyEvent` (línea 54) para determinar si la tecla en el evento fue una tecla de acción. Además, se hace una llamada al método `getModifiers` de `InputEvent` (línea 57) para determinar si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla. El resultado de este método se pasa al método `getKeyModifiersText` de `KeyEvent`, el cual produce una cadena que contiene los nombres de las teclas modificadoras que se oprimieron.

(*Nota:* Si necesita probar una tecla específica en el teclado, la clase `KeyEvent` proporciona una *constante de tecla* para cada tecla del teclado. Estas constantes pueden utilizarse desde los manejadores de eventos de teclas para determinar si se oprimió una tecla específica. Además, para determinar si las teclas *Alt*, *Ctrl*, *Meta* y *Mayús* se oprimen individualmente, cada uno de los métodos `isAltDown`, `isControlDown`, `isMetaDown` e `isShiftDown` devuelven un valor `boolean`, indicando si se oprimió dicha tecla durante el evento clave.)

13.15 Administradores de esquemas

Los *administradores de esquemas* se proporcionan para ordenar los componentes de la GUI en un contenedor, para fines de presentación; además, proporcionan herramientas básicas de distribución visual, que son más fáciles de usar que determinar la posición y tamaño exactos de cada componente de la GUI. Esta funcionalidad permite al programador concentrarse en la vista y sentido básicos, y deja que el administrador de esquemas procese la mayoría de los detalles de la distribución visual.

Observación de apariencia visual 13.9



La mayoría de los entornos de programación de Java proporcionan herramientas de diseño de la GUI, las cuales ayudan a un programador a diseñar gráficamente una GUI; después las herramientas de diseño escriben código en Java para crear la GUI.

Algunos diseñadores de la GUI también permiten al programador utilizar los administradores de esquemas que se describen en éste y en el capítulo 14. En la figura 13.23 se sintetizan los administradores de esquema que se presentan en este capítulo. Los demás administradores de esquemas los veremos en el capítulo 14.

La mayoría de los ejemplos de subprogramas y aplicaciones anteriores en los que creamos nuestra propia GUI, utilizan el administrador de esquemas `FlowLayout`. La clase `FlowLayout` hereda de la clase `Object` e implementa a la interfaz `LayoutManager`, la cual declara los métodos que utiliza un administrador de esquemas para ordenar y ajustar el tamaño de los componentes de la GUI en un contenedor.

13.15.1 FlowLayout

Éste es el administrador de esquemas más simple. Los componentes de la GUI se colocan en un contenedor, de izquierda a derecha, en el orden en el que se agregaron al contenedor. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. La clase `FlowLayout` permite a los componentes de la GUI *alinearse a la izquierda*, *al centro* (el valor predeterminado) y *a la derecha*.

La aplicación de la figura 13.24 crea tres objetos `JButton` y los agrega a la aplicación, utilizando un administrador de esquemas `FlowLayout`. Los componentes se alinean hacia el centro de manera predeterminada. Cuando el usuario hace clic en **Izquierda**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la izquierda. Cuando el usuario hace clic en **Derecha**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la derecha. Cuando el usuario hace clic en **Centro**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado hacia el centro. Cada botón tiene su propio manejador de eventos que se declara con una clase interna, la cual implementa a `ActionListener`. Las ventanas de salida de ejemplo muestran cada una de las alineaciones de `FlowLayout`. Además, la última ventana de salida de ejemplo muestra la alineación centrada después de ajustar el tamaño de la ventana a una anchura menor. Observe que el botón **Derecha** fluye hacia una nueva línea.

Administrador de esquemas	Descripción
`FlowLayout`	Es el predeterminado para `java.awt.Applet`, `java.awt.Panel` y `javax.swing.JPanel`. Coloca los componentes secuencialmente (de izquierda a derecha) en el orden en que se agregaron. También es posible especificar el orden de los componentes utilizando el método `add` de `Container`, el cual toma un objeto `Component` y una posición de índice entero como argumentos.
`BorderLayout`	Es el predeterminado para los paneles de contenido de objetos `JFrame` (y otras ventanas) y `JApplet`. Ordena los componentes en cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`.
`GridLayout`	Ordena los componentes en filas y columnas.

Figura 13.23 Administradores de esquemas.

http://librosysolucionarios.net

```
1  // Fig. 13.24: DemoFlowLayout.java
2  // Demostración de las alineaciones de FlowLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoFlowLayout extends JFrame {
8     private JButton botonIzquierda, botonCentro, botonDerecho;
9     private Container contenedor;
10    private FlowLayout esquema;
11
12    // configurar la GUI y registrar componentes de escucha de los botones
13    public DemoFlowLayout()
14    {
15       super( "Demostración de FlowLayout" );
16
17       esquema = new FlowLayout();
18
19       // obtener panel de contenido y establecer su esquema
20       contenedor = getContentPane();
21       contenedor.setLayout( esquema );
22
23       // establecer objeto botonIzquierda y registrar componente de escucha
24       botonIzquierda = new JButton( "Izquierda" );
25       contenedor.add( botonIzquierda );
26       botonIzquierda.addActionListener(
27
28       new ActionListener() {  // clase interna anónima
29
30             // procesar evento de botonIzquierda
31             public void actionPerformed( ActionEvent evento )
32             {
33                esquema.setAlignment( FlowLayout.LEFT );
34
35                // realinear componentes adjuntos
36                esquema.layoutContainer( contenedor );
37             }
38
39          } // fin de la clase interna anónima
40
41       ); // fin de la llamada a addActionListener
42
43       // establecer objeto botonCentro y registrar componente de escucha
44       botonCentro = new JButton( "Centro" );
45       contenedor.add( botonCentro );
46       botonCentro.addActionListener(
47
48          new ActionListener() {  // clase interna anónima
49
50             // procesar evento de botonCentro
51             public void actionPerformed( ActionEvent evento )
52             {
53                esquema.setAlignment( FlowLayout.CENTER );
54
55                // realinear componentes adjuntos
```

Figura 13.24 `FlowLayout` permite a los componentes fluir a través de varias líneas. (Parte 1 de 2.)


```
56                esquema.layoutContainer( contenedor );
57             }
58          }
59       );
60
61       // establecer objeto botonDerecho y registrar componente de escucha
62       botonDerecho = new JButton( "Derecha" );
63       contenedor.add( botonDerecho );
64       botonDerecho.addActionListener(
65
66          new ActionListener() {  // clase interna anónima
67
68             // procesar evento de botonDerecho
69             public void actionPerformed( ActionEvent evento )
70             {
71                esquema.setAlignment( FlowLayout.RIGHT );
72
73                // realinear componentes adjuntos
74                esquema.layoutContainer( contenedor );
75             }
76          }
77       );
78
79       setSize( 300, 75 );
80       setVisible( true );
81
82    } // fin del constructor DemoFlowLayout
83
84    public static void main( String args[] )
85    {
86       DemoFlowLayout aplicacion = new DemoFlowLayout();
87       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
88    }
89
90 } // fin de la clase DemoFlowLayout
```

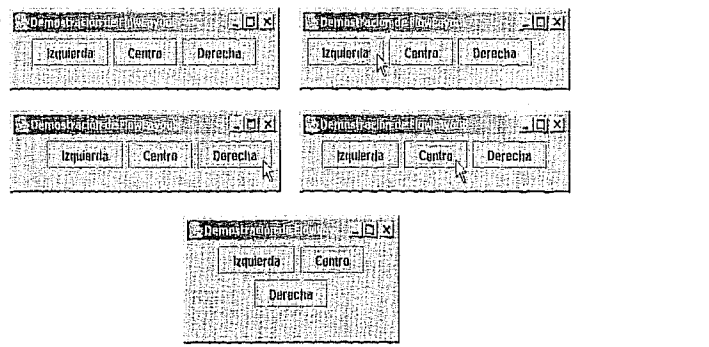


Figura 13.24 `FlowLayout` permite a los componentes fluir a través de varias líneas. (Parte 2 de 2.)

Como se vio anteriormente, el esquema de un contenedor se establece mediante el método ***`setLayout`*** de la clase `Container`. En la línea 21 se establece el administrador de esquemas del panel de contenido en `FlowLayout`, el cual se declara en la línea 17. Generalmente, el esquema se establece antes de agregar cualquier componente de la GUI a un contenedor.

http://librosysolucionarios.net



Observación de apariencia visual 13.10

Cada contenedor puede tener solamente un administrador de esquemas a la vez. Varios contenedores en el mismo programa pueden tener distintos administradores de esquemas.



Observación de apariencia visual 13.11

En un objeto `Container` *sin administrador de esquema, el programador debe colocar los widgets contenidos en dicho contenedor, y debe encargarse de que, en los eventos de ajuste de tamaño, todos los componentes se reposicionen correctamente.*

El manejador de eventos `actionPerformed` de cada botón ejecuta dos instrucciones. Por ejemplo, en la línea 33 en el método `actionPerformed` para el botón `izquierda` se utiliza el método `setAlignment` de `FlowLayout` para cambiar la alineación del objeto `FlowLayout` a la izquierda (`FlowLayout.LEFT`). En la línea 36 se utiliza el método `layoutContainer` de la interfaz `LayoutManager` para especificar que el panel de contenido debe reordenarse, con base en el esquema ajustado.

Dependiendo de con cuál botón se haya hecho clic, el método `actionPerformed` de cada botón establecerá la alineación de `FlowLayout` a `LEFT` (línea 33), `CENTER` (línea 53) o `RIGHT` (línea 71).

13.15.2 BorderLayout

El administrador de esquemas `BorderLayout` (el predeterminado para el panel de contenido de un objeto `JFrame` o `JApplet`) ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. `NORTH` corresponde a la parte superior del contenedor. La clase `BorderLayout` extiende a `Object` e implementa a la interfaz `LayoutManager2` (una subinterfaz de `LayoutManager`, que agrega varios métodos para un mejor procesamiento de los esquemas).

En `BorderLayout` pueden mostrarse hasta cinco componentes; uno en cada región. El componente que se coloca en cada región puede ser un contenedor, al cual se pueden adjuntar otros componentes. Los componentes que se colocan en las regiones `NORTH` y `SOUTH` se extienden horizontalmente hacia los lados del contenedor, y tienen la misma altura que los componentes que se colocan en esas regiones. Las regiones `EAST` y `WEST` se expanden verticalmente entre las regiones `NORTH` y `SOUTH`, y tienen la misma anchura que los componentes que se coloquen dentro de ellas. El componente que se coloca en la región `CENTER` se expande para rellenar todo el espacio restante en el esquema (esto explica por qué el objeto `JTextArea` de la figura 13.22 ocupa toda la ventana). Si las cinco regiones están ocupadas, todo el espacio del contenedor se cubre con los componentes de la GUI. Si las regiones `NORTH` o `SOUTH` no están ocupadas, los componentes de la GUI en las regiones `EAST`, `CENTER` y `WEST` se expanden verticalmente para rellenar el espacio restante. Si las regiones `EAST` o `WEST` no están ocupadas, el componente de la GUI en la región `CENTER` se expande horizontalmente para rellenar el espacio restante. Si la región `CENTER` no está ocupada, el área se deja vacía; los demás componentes de la GUI no se expanden para rellenar el espacio restante. La aplicación de la figura 13.25 demuestra el administrador de esquemas `BorderLayout`, utilizando cinco objetos `JButton`.

```
1  // Fig. 13.25: DemoBorderLayout.java
2  // Demostración de BorderLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoBorderLayout extends JFrame implements ActionListener {
8     private JButton botones[];
9     private final String nombres[] = { "Ocultar norte", "Ocultar sur",
10       "Ocultar este", "Ocultar oeste", "Ocultar centro" };
11    private BorderLayout esquema;
12
13    // configurar la GUI y el manejo de eventos
```

Figura 13.25 `BorderLayout` con cinco botones. (Parte 1 de 3.)


```
14    public DemoBorderLayout()
15    {
16       super( "Demostración de BorderLayout" );
17
18       esquema = new BorderLayout( 5, 5 ); // espacios libres de 5 píxeles
19
20       // obtener panel de contenido y establecer su esquema
21       Container contenedor = getContentPane();
22       contenedor.setLayout( esquema );
23
24       // instanciar objetos botón
25       botones = new JButton[ nombres.length ];
26
27       for ( int cuenta = 0; cuenta < nombres.length; cuenta++ ) {
28          botones[ cuenta ] = new JButton( nombres[ cuenta ] );
29          botones[ cuenta ].addActionListener( this );
30       }
31
32       // colocar botones en BorderLayout; no importa el orden
33       contenedor.add( botones[ 0 ], BorderLayout.NORTH );
34       contenedor.add( botones[ 1 ], BorderLayout.SOUTH );
35       contenedor.add( botones[ 2 ], BorderLayout.EAST );
36       contenedor.add( botones[ 3 ], BorderLayout.WEST );
37       contenedor.add( botones[ 4 ], BorderLayout.CENTER );
38
39       setSize( 300, 200 );
40       setVisible( true );
41
42    } // fin del constructor de DemoBorderLayout
43
44    // manejar eventos de botón
45    public void actionPerformed( ActionEvent evento )
46    {
47       for ( int cuenta = 0; cuenta < botones.length; cuenta++ )
48
49          if ( evento.getSource() == botones[ cuenta ] )
50             botones[ cuenta ].setVisible( false );
51          else
52             botones[ cuenta ].setVisible( true );
53
54       // re-esquematizar el panel de contenido
55       esquema.layoutContainer( getContentPane() );
56    }
57
58    public static void main( String args[] )
59    {
60       DemoBorderLayout aplicacion = new DemoBorderLayout();
61       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
62    }
63
64 } // fin de la clase DemoBorderLayout
```

Figura 13.25 `BorderLayout` con cinco botones. (Parte 1 de 3.)

http://librosysolucionarios.net

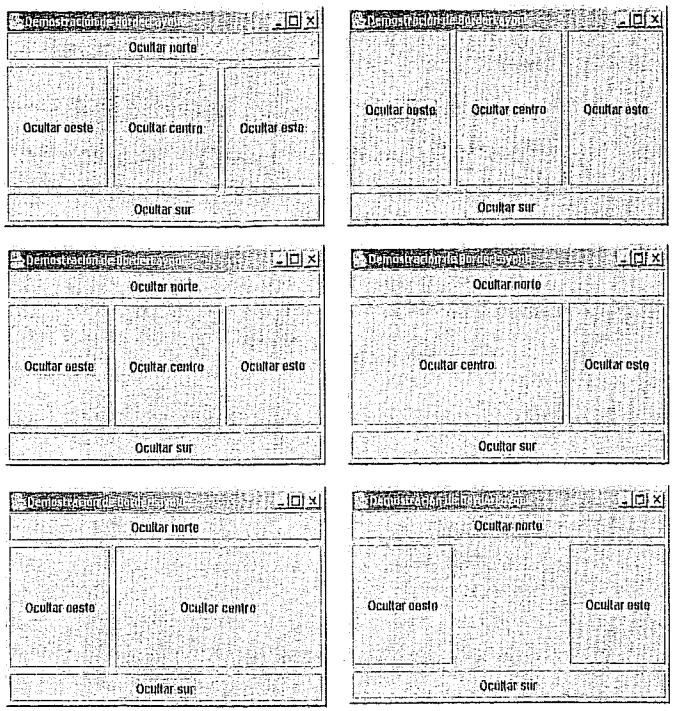


Figura 13.25 `BorderLayout` con cinco botones. (Parte 3 de 3.)

En la línea 18 se crea un objeto `BorderLayout`. Los argumentos del constructor especifican el número de píxeles entre los componentes que se ordenan en forma horizontal (*espacio libre horizontal*) y el número de píxeles entre los componentes que se ordenan en forma vertical (*espacio libre vertical*), respectivamente. El valor predeterminado es un píxel de espacio libre horizontal y vertical. En la línea 22 se utiliza el método `setLayout` para establecer el esquema del panel de contenido en `esquema`.

Agregamos objetos `Component` a un objeto `BorderLayout` con otra versión del método `add` de `Container` que toma dos argumentos: el objeto `Component` que se va a agregar y la región en la que debe aparecer este objeto. Por ejemplo, en la línea 33 se especifica que `botones[ 0 ]` debe aparecer en la región `NORTH`. Los componentes pueden agregarse en cualquier orden, pero sólo debe agregarse un componente a cada región.

Observación de apariencia visual 13.12

Si no se especifica una región al agregar un objeto `Component` a un objeto `BorderLayout`, el administrador de esquemas asume que el objeto `Component` debe agregarse a la región `BorderLayout.CENTER`.

Error común de programación 13.5

Cuando se agrega más de un componente a una región en un objeto `BorderLayout`, sólo se mostrará el último componente agregado a esa región. No hay un error que indique este problema.

Cuando el usuario hace clic en un objeto `JButton` específico en el esquema, se ejecuta el método `actionPerformed` (líneas 45 a 56). El ciclo `for` de las líneas 47 a 52 utiliza una instrucción `if...else` para ocultar el objeto `JButton` específico que generó el evento. Se hace una llamada al método ***`setVisible`*** (heredado en `JButton` de la clase `Component`) con un argumento `false` (línea 50) para ocultar el objeto `JButton`. Si el objeto `JButton` actual en el arreglo no es el que generó el evento, [illegible] llamada al método `setVisible` con un argumento `true` (línea 52) para asegurar que se muestre ese objeto `JButton`

http://librosysolucionarios.net

en la pantalla. En la línea 55 se utiliza el método `layoutContainer` de `LayoutManager` para recalcular el esquema del panel de contenido. Observe en las capturas de pantalla de la figura 13.25 que ciertas regiones en el objeto `BorderLayout` cambian su forma a medida que se ocultan y vuelven a mostrar los objetos `JButton` en las otras regiones. Pruebe ajustando el tamaño de la ventana para ver cómo se ajusta el tamaño de las diversas regiones, con base en la anchura y altura de la ventana. Para obtener esquemas más complejos, puede agrupar componentes en objetos `JPanel`, cada uno con un administrador de esquemas separado. Coloque los objetos `JPanel` en el objeto `JFrame`, utilizando ya sea el esquema `BorderLayout` predeterminado, o algún otro esquema.

13.15.3 GridLayout

El administrador de esquemas ***`GridLayout`*** divide el contenedor en una cuadrícula, de manera que los componentes puedan colocarse en filas y columnas. La clase `GridLayout` hereda directamente de la clase `Object` e implementa a la interfaz `LayoutManager`. Todo objeto `Component` en un objeto `GridLayout` tiene la misma anchura y altura. Los componentes se agregan a un objeto `GridLayout` empezando en la celda superior izquierda de la cuadrícula, y procediendo de izquierda a derecha hasta que la fila esté llena. Después el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente. En la figura 13.26 se demuestra el administrador de esquemas `GridLayout`, utilizando seis objetos `JButton`.

```
1  // Fig. 13.26: DemoGridLayout.java
2  // Demostración de GridLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoGridLayout extends JFrame implements ActionListener {
8     private JButton botones[];
9     private final String nombres[] =
10       { "uno", "dos", "tres", "cuatro", "cinco", "seis" };
11    private boolean alternar = true;
12    private Container contenedor;
13    private GridLayout cuadricula1, cuadricula2;
14
15    // configurar la GUI
16    public DemoGridLayout()
17    {
18       super( "Demostración de GridLayout" );
19
20       // establecer esquemas
21       cuadricula1 = new GridLayout( 2, 3, 5, 5 );
22       cuadricula2 = new GridLayout( 3, 2 );
23
24       // obtener panel de contenido y establecer su esquema
25       contenedor = getContentPane();
26       contenedor.setLayout( cuadricula1 );
27
28       // crear y agregar botones
29       botones = new JButton[ nombres.length ];
30
31       for ( int cuenta = 0; cuenta < nombres.length; cuenta++ ) {
32          botones[ cuenta ] = new JButton( nombres[ cuenta ] );
33          botones[ cuenta ].addActionListener( this );
34          contenedor.add( botones[ cuenta ] );
```

Figura 13.26 `GridLayout` con seis botones. (Parte 1 de 2.)

```
35        }
36
37        setSize( 300, 150 );
38        setVisible( true );
39
40     } // fin del constructor de DemoGridLayout
41
42     // manejar eventos de botón, alternando entre los esquemas
43     public void actionPerformed( ActionEvent evento )
44     {
45        if ( alternar )
46           contenedor.setLayout( cuadricula2 );
47        else
48           contenedor.setLayout( cuadricula1 );
49
50        alternar = !alternar;  // establecer alternar en el valor opuesto
51        contenedor.validate();
52     }
53
54     public static void main( String args[] )
55     {
56        DemoGridLayout aplicacion = new DemoGridLayout();
57        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
58     }
59
60  } // fin de la clase DemoGridLayout
```

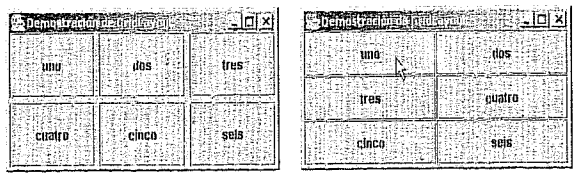


Figura 13.26 `GridLayout` con seis botones. (Parte 2 de 2.)

En las líneas 21 y 22 se crean dos objetos `GridLayout` (las variables `cuadricula1` y `cuadricula2` se declaran en la línea 13). El constructor de `GridLayout` que se utiliza en la línea 21 especifica un objeto `GridLayout` con 2 filas, 3 columnas, 5 píxeles de espacio libre horizontal entre objetos `Component` en la cuadrícula y 5 píxeles de espacio libre vertical entre objetos `Component` en la cuadrícula. El constructor de `GridLayout` que se utiliza en la línea 22 especifica un objeto `GridLayout` con 3 filas, 2 columnas y sin espacio libre.

Los objetos `JButton` en este ejemplo se ordenan inicialmente utilizando `cuadricula1` (que se establece para el panel de contenido en la línea 26, mediante el método `setLayout`). El primer componente se agrega a la primera columna de la primera fila. El siguiente componente se agrega a la segunda columna de la primera fila, y así sucesivamente. Cuando se oprime un objeto `JButton`, se hace una llamada al método `actionPerformed` (líneas 43 a 52). Todas las llamadas a `actionPerformed` alternan el esquema entre `cuadricula2` y `cuadricula1`, utilizando la variable `boolean` llamada `alternar` para determinar el siguiente esquema a establecer.

En la línea 51 se muestra otra manera para cambiar el formato a un contenedor para el cual haya cambiado el esquema. El método ***validate*** de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI que se muestran en pantalla.

13.16 Paneles

Las GUIs complejas (como la de la figura 13.1) requieren que cada componente se coloque en una ubicación exacta. A menudo consisten de varios *paneles*, en donde los componentes de cada panel se ordenan en un esquema específico. Los paneles se crean con la clase ***JPanel***, una subclase de `JComponent`. La clase `JComponent` extiende a la clase `java.awt.Container`, por lo que todo `JPanel` es un `Container`. Por lo tanto, se pueden agregar componentes a los objetos `JPanel`, incluyendo otros paneles. El programa de la figura 13.27 demuestra cómo puede usarse un objeto `JPanel` para crear un esquema más complejo para objetos `Component`.

```
1  // Fig. 13.27: DemoPanel.java
2  // Uso de un objeto JPanel para ayudar a distribuir los componentes.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoPanel extends JFrame {
8     private JPanel panelBotones;
9     private JButton botones[];
10
11    // configurar la GUI
12    public DemoPanel()
13    {
14       super( "Demostración JPanel" );
15
16       // obtener panel de contenido
17       Container contenedor = getContentPane();
18
19       // crear arreglo botones
20       botones = new JButton[ 5 ];
21
22       // configurar panel y establecer su esquema
23       panelBotones = new JPanel();
24       panelBotones.setLayout( new GridLayout( 1, botones.length ) );
25
26       // crear y agregar botones
27       for ( int cuenta = 0; cuenta < botones.length; cuenta++ ) {
28          botones[ cuenta ] = new JButton( "Botón " + ( cuenta + 1 ) );
29          panelBotones.add( botones[ cuenta ] );
30       }
31
32       contenedor.add( panelBotones, BorderLayout.SOUTH );
33
34       setSize( 425, 150 );
35       setVisible( true );
36
37    } // fin del constructor de DemoPanel
38
39    public static void main( String args[] )
40    {
41       DemoPanel aplicacion = new DemoPanel();
42       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
43    }
```

Figura 13.27 `JPanel` con cinco objetos `JButton` en un objeto `GridLayout`, adjunto a la región `SOUTH` de un objeto `BorderLayout`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
44
45 } // fin de la clase DemoPanel
```

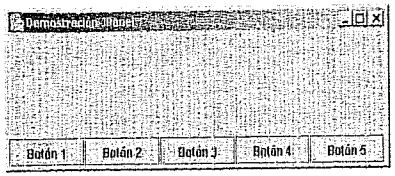


Figura 13.27 `JPanel` con cinco objetos `JButton` en un objeto `GridLayout`, adjunto a la región `SOUTH` de un objeto `BorderLayout`. (Parte 2 de 2.)

Una vez declarado el objeto `JPanel` llamado `panelBotones` en la línea 8, y creado en la línea 23, en la línea 24 se establece el esquema de `panelBotones` en `GridLayout` con una fila y cinco columnas (hay cinco objetos `JButton` en el arreglo `botones`). En las líneas 27 a 30 se agregan los cinco objetos `JButton` del arreglo `botones` al objeto `JPanel` en el ciclo. En la línea 29 se agregan los botones directamente al objeto `JPanel` (la clase `JPanel` no tiene un panel de contenido, a diferencia de `JApplet` o `JFrame`). En la línea 32 se utiliza el objeto `BorderLayout` predeterminado del panel de contenido para agregar `panelBotones` a la región `SOUTH`. Observe que esta región tiene la misma altura que los botones en `panelBotones`. Un objeto `JPanel` ajusta su tamaño de acuerdo con los componentes que contiene. A medida que se agregan más componentes, el objeto `JPanel` crece (de acuerdo con las restricciones de su administrador de esquemas) para dar cabida a esos nuevos componentes. Ajuste el tamaño de la ventana para que vea cómo el administrador de esquemas afecta al tamaño de los objetos `JButton`.

13.17 (Ejemplo práctico opcional) Acerca de los objetos: Casos de uso

Las secciones "Acerca de los objetos" de capítulos anteriores se han concentrado en el modelo de la simulación del elevador. Hemos identificado y agudizado la estructura y el comportamiento de nuestro sistema. En esta sección modelaremos la interacción entre el usuario y nuestra simulación de elevador, a través del diagrama de caso-uso de UML, el cual describe el conjunto de escenarios que ocurren entre el usuario y el sistema.

Diagramas de caso-uso

Cuando los desarrolladores empiezan un proyecto, raras veces comienzan con un enunciado del problema tan detallado como el que proporcionamos en la sección 2.9. Este documento y los demás son el resultado de la fase de recopilación de requerimientos. En esta fase usted debe reunirse con las personas que desean que usted construya un sistema, y con las personas que eventualmente utilizarán ese sistema. Debe utilizar la información obtenida en estas reuniones para compilar una lista de *requerimientos del sistema*. Estos requerimientos lo guiarán a usted y a sus compañeros de desarrollo, a medida que analizan y diseñan el sistema. En nuestro ejemplo práctico, el enunciado del problema describe los requerimientos de nuestra simulación de elevador con suficiente detalle como para que usted no necesite pasar por una extensa fase de análisis. Esta fase es sumamente importante; es conveniente que consulte las referencias que proporcionamos en la sección 2.9 para que aprenda más acerca del análisis orientado a objetos.

UML proporciona el *diagrama de caso-uso* para facilitar la recopilación de los requerimientos. Este diagrama modela las interacciones entre los clientes externos del sistema y los *casos de uso* del mismo. Cada caso de uso representa una herramienta distinta que el sistema proporciona a sus clientes. Por ejemplo, las máquinas de cajero automático generalmente tienen varios casos de uso, incluyendo "Depositar dinero", "Retirar dinero" y "Transferir fondos".

En sistemas más grandes, los diagramas de caso-uso son herramientas indispensables que ayudan a los diseñadores de sistemas a mantenerse enfocados en satisfacer las necesidades de los usuarios. El objetivo del diagrama de caso-uso es mostrar los tipos de interacciones que tienen los usuarios con un sistema, sin tener que

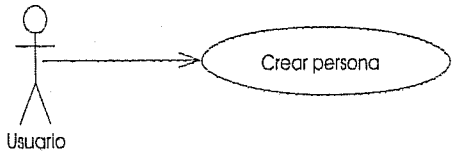


Figura 13.28 Diagrama de caso-uso para la simulación del elevador, desde la perspectiva del usuario.

proporcionar los detalles de esas interacciones; desde luego que esos detalles se proporcionan en otros diagramas de UML. A menudo, los diagramas de caso-uso vienen acompañados de texto informal que describe las interacciones con más detalle.

En la figura 13.28 se muestra el diagrama de caso-uso para nuestra simulación de elevador. La figura lineal representa a un *actor*, el cual a su vez representa a un conjunto de roles que puede desempeñar una *entidad externa* (como una persona u otro sistema). Considere de nuevo nuestro ejemplo de la máquina de cajero automático (ATM). El actor es un `ClienteBanco` que puede depositar, retirar y transferir fondos del ATM. En este sentido, `ClienteBanco` es más como una clase que como un objeto; no es una persona real, sino que describe los roles que una persona real (cuando desempeña el papel de un `ClienteBanco`) puede desempeñar al interactuar con el ATM (depositar, retirar y transferir fondos). Una persona es una entidad externa que puede desempeñar el papel de un `ClienteBanco`. De la misma forma que un objeto es una instancia de una clase, una persona que desempeña el papel de un `ClienteBanco` que lleva a cabo uno de sus roles (como hacer un depósito) es una *instancia del actor* `ClienteBanco`. Por ejemplo, cuando una persona llamada María desempeña el papel de un `ClienteBanco` que hace un depósito, María (en el papel del depositante) se convierte en una instancia del actor `ClienteBanco`. Posteriormente en ese día, otra persona llamada Juan puede ser otra instancia del actor `ClienteBanco`. En el transcurso del día, varios cientos de personas podrían utilizar la máquina ATM; algunos podrían ser "clientes depositantes", otros "clientes retiradores" y otros más "clientes transferidores", pero todas estas personas son instancias del actor `ClienteBanco`.

El enunciado del problema en nuestra simulación de elevador suministra los actores: "El usuario requiere la habilidad de crear una persona en la simulación, y situarla en un piso dado". Por lo tanto, el actor en este caso de uso es el usuario que controla la simulación (es decir, el usuario que hace clic en los botones para crear nuevos objetos `Persona` en la simulación). Una entidad externa (una persona real) desempeña el papel del usuario para controlar la situación. En nuestro sistema, el caso de uso es "Crear persona", que consta de crear un objeto `Persona` y después colocar ese objeto en el primer o segundo `Piso`. En la figura 13.28 se modela nuestro actor llamado "Usuario". La clase del actor aparece debajo de éste. UML modela cada caso de uso como un óvalo.

Hay una vista alterna razonable del caso de uso de nuestra simulación de elevador. El enunciado del problema de la sección 2.9 mencionaba que la compañía solicitó la simulación de elevador para "determinar si el elevador cumplirá con las necesidades de la compañía". Diseñaremos una simulación de un escenario del mundo real: el objeto `Persona` en la simulación representa a un humano real, utilizando un elevador real. Por lo tanto, especificar un caso de uso desde la perspectiva del objeto `Persona` nos ayuda a modelar la manera en que una persona real utiliza un sistema de elevador real. Ofrecemos el caso de uso de la figura 13.29, titulado "Reubicar persona". Este caso de uso describe a la `Persona` que se mueve (reubica) hacia el otro `Piso`. (La `Persona` viaja al segundo `Piso` si empieza en el primer `Piso`, y viceversa.) Este caso de uso abarca todas las acciones que realiza la `Persona` a lo largo de su recorrido, como caminar a través de un `Piso` hacia el `Elevador`, oprimir un `Boton` y viajar en el `Elevador` hacia el otro `Piso`.

Debemos asegurarnos que nuestros casos de uso no modelen interacciones que sean demasiado específicas entre el cliente externo y el sistema. Por ejemplo, no subdividimos cada caso de uso en dos casos de uso separados (por ejemplo, "Crear persona en primer Piso" y "Crear Persona en segundo Piso", o "Reubicar persona a primer Piso" y "Reubicar persona a segundo Piso") ya que la funcionalidad de dichos casos de uso es repetitiva (es decir, estos casos de uso aparentemente alternativos son realmente el mismo). Una subdivisión incorrecta y repetitiva de los casos de uso podría crear problemas durante la implementación. Por ejemplo, si el diseñador de una máquina de cajero automático separara su caso de uso "Retirar dinero" en casos de uso "Retirar cantidades específicas" (por ejemplo, "Retirar $1.00", "Retirar $2.00", etcétera), podría existir un enorme nú-

http://librosysolucionarios.net

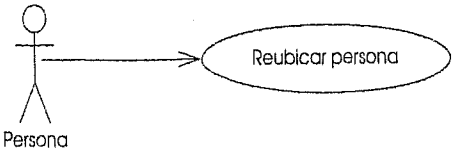


Figura 13.29 Diagrama de caso-uso, desde la perspectiva de una `Persona`.

mero de casos de uso para el sistema. Esto haría tedioso el proceso de entender los requerimientos. (Nuestro sistema de elevador contiene sólo dos pisos, por lo que separar el caso de uso en dos no ocasionaría tanto trabajo adicional; no obstante, si nuestro sistema tuviera 100 pisos, sería muy difícil crear 100 casos de uso.)

RESUMEN

- Una GUI (interfaz gráfica de usuario) presenta una interfaz ilustrada para un programa. Una GUI proporciona a un programa una apariencia visual única.
- Al proporcionar a distintas aplicaciones un conjunto consistente de componentes de interfaz de usuario intuitivos, las GUIs permiten al usuario invertir más tiempo en aprender a utilizar el programa de una forma productiva.
- Las GUIs se crean a partir de componentes de la GUI (a los que algunas veces se les conoce como controles o widgets). Un componente de la GUI es un objeto visual con el que el usuario interactúa mediante el ratón, teclado o cualquier otra forma de entrada.
- Los componentes de la GUI de Swing se declaran en el paquete `javax.swing`. La mayoría de los componentes de Swing están escritos, se manipulan y muestran completamente en Java. A dichos componentes se les conoce como componentes ligeros de la GUI. Swing forma parte de JFC (Java Foundation Classes).
- Los componentes originales de la GUI del paquete `java.awt` de AWT (Abstract Windows Toolkit) están enlazados directamente con las herramientas de interfaz de usuario de la plataforma local.
- A los componentes de AWT se les llama componentes pesados; deben depender del sistema de ventanas de la plataforma local para determinar su funcionalidad y su apariencia visual.
- Varios componentes de la GUI de Swing son componentes pesados de la GUI; en especial, las subclases de `java.awt.Window` (como `JFrame`), que muestran ventanas en la pantalla. Los componentes pesados de la GUI de Swing son menos flexibles que los componentes ligeros.
- Gran parte de la funcionalidad de cada componente de la GUI de Swing se hereda de las clases `Component`, `Container` y `JComponent` (una superclase de la mayoría de los componentes de Swing).
- Un objeto `Container` organiza un conjunto de componentes de la GUI.
- Los objetos `JLabel` proporcionan instrucciones de texto o información en una GUI.
- El método `setToolTipText` de `JComponent` especifica la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre un objeto `JComponent` en la GUI.
- Muchos componentes de Swing pueden mostrar imágenes especificando un objeto `Icon` como argumento para su constructor, o utilizando un método llamado `setIcon`.
- La clase `ImageIcon` soporta varios formatos de imagen, incluyendo PNG (Gráficos portables de red), GIF (Formato de intercambio de gráficos) y JPEG (Grupo unido de expertos en fotografía).
- La interfaz `SwingConstants` (paquete `javax.swing`) declara un conjunto de constantes enteras comunes que se utilizan con muchos componentes de Swing.
- De manera predeterminada, el texto de un objeto `JComponent` aparece a la derecha de la imagen, cuando este objeto contiene tanto texto como una imagen.
- Las alineaciones vertical y horizontal de un objeto `JLabel` pueden establecerse mediante los métodos `setHorizontalAlignment` y `setVerticalAlignment`. El método `setText` establece el texto a mostrar en la etiqueta. El método `getText` recupera el texto actual que se muestra en una etiqueta. Los métodos `setHorizontalTextPosition` y `setVerticalTextPosition` especifican la posición del texto en una etiqueta.
- El método `setIcon` de la clase `JComponent` establece el objeto `Icon` a mostrar en un objeto `JComponent`. El método `getIcon` recupera el objeto `Icon` actual que se muestra en un objeto `JComponent`.

- Las GUIs generan eventos cuando el usuario interactúa con ellas. La información acerca de un evento de la GUI se almacena en un objeto de una clase que extiende a `AWTEvent`.
- Para procesar un evento, el programador debe registrar un componente de escucha de eventos e implementar uno o más manejadores de eventos.
- El uso de componentes de escucha de eventos en el manejo de eventos se conoce como el modelo de delegación de eventos: el procesamiento de un evento se delega a un objeto específico en el programa.
- Cuando ocurre un evento, el componente de la GUI con el que interactuó el usuario notifica a sus componentes de escucha registrados, llamando al método manejador de eventos apropiado de cada componente de escucha.
- Los objetos `JTextField` y `JPasswordField` son áreas de una sola línea en las que el usuario puede introducir texto desde el teclado, o también se usan solamente para mostrar texto. Un objeto `JPasswordField` muestra que los caracteres se están escribiendo a medida que el usuario los introduce, pero oculta los caracteres.
- Cuando el usuario introduce datos en un objeto `JTextField` o `JPasswordField` y oprime la tecla *Intro*, ocurre un evento `ActionEvent`.
- El método `setEditable` de `JTextComponent` determina si el usuario puede modificar el texto en un objeto `JTextComponent`.
- El método `getPassword` de `JPasswordField` devuelve la contraseña como un arreglo de tipo `char`.
- Todo objeto `JComponent` contiene un objeto de la clase `EventListenerList`, en donde se almacenan todos los componentes de escucha registrados.
- Todo objeto `JComponent` soporta varios tipos de eventos distintos, incluyendo eventos de ratón, de teclas y otros más. Cuando ocurre un evento, éste se despacha solamente a los componentes de escucha de eventos registrados del tipo apropiado. Cada tipo de evento tiene su correspondiente interfaz de escucha de eventos.
- Un objeto `JButton` genera un evento `ActionEvent` cuando el usuario hace clic en el botón con el ratón.
- Un objeto `AbstractButton` puede tener un objeto `Icon` de sustitución, que se muestre cuando el cursor del ratón esté colocado sobre ese objeto botón. El icono cambia a medida que el cursor del ratón se mueve hacia dentro y hacia fuera del área del botón en la pantalla. El método `setRolloverIcon` de `AbstractButton` especifica el icono de sustitución.
- Hay tres tipos de botones de estado: `JToggleButton`, `JCheckBox` y `JRadioButton`, con valores encendido/apagado o verdadero/falso (true/false). `JCheckBox` y `JRadioButton` son subclases de `JToggleButton`.
- Cuando el usuario hace clic en un objeto `JCheckBox`, se genera un evento `ItemEvent` que puede manejarse mediante un objeto `ItemListener`. Los objetos `ItemListener` deben declarar el método `itemStateChanged`. El método `getStateChange` de `ItemEvent` determina el estado de un objeto `JToggleButton`.
- Los objetos `JRadioButton` son similares a los objetos `JCheckBox` en cuanto a que tienen dos estados: seleccionado y no seleccionado. Los objetos `JRadioButton` normalmente aparecen como un grupo, en el cual sólo puede seleccionarse un botón de opción a la vez. Los objetos `JRadioButton` generan eventos `ItemEvent` cuando se hace clic en ellos.
- La relación lógica entre los botones de opción se mantiene mediante un objeto `ButtonGroup`.
- El constructor de `JRadioButton` proporciona la etiqueta que aparece a la derecha del objeto `JRadioButton` de manera predeterminada, así como el estado inicial de este objeto. Un segundo argumento `true` indica que el objeto `JRadioButton` debe aparecer seleccionado al mostrarse en pantalla.
- El método `add` de `ButtonGroup` asocia un objeto `JRadioButton` con un objeto `ButtonGroup`. Si se agrega al grupo más de un objeto `JRadioButton` seleccionado, el primer objeto `JRadioButton` seleccionado que se haya agregado será el que quede seleccionado cuando se muestre la GUI.
- Un objeto `JComboBox` (conocido también como lista desplegable) proporciona una lista de elementos de los cuales el usuario puede seleccionar sólo uno a la vez. Los objetos `JComboBox` generan eventos `ItemEvent`. El primer elemento se agrega a un objeto `JComboBox` en el índice 0, el siguiente elemento se agrega en el índice 1, etcétera. El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado cuando se muestra el objeto `JComboBox` en pantalla. El método `getSelectedIndex` de `JComboBox` devuelve el índice del elemento seleccionado.
- Un objeto `JList` muestra una serie de elementos de los cuales el usuario puede seleccionar uno o más a la vez. La clase `JList` soporta las listas de selección simple y múltiple. Cuando se hace clic en un elemento de un objeto `JList`, ocurre un evento `ListSelectionEvent`.
- El método `setVisibleRowCount` de `JList` determina el número de elementos que estarán visibles en la lista. El método `getSelectionMode` especifica el modo de selección para esa lista.

http://librosysolucionarios.net

- Una lista `SINGLE_INTERVAL_SELECTION` permite la selección de un rango contiguo de elementos. Para ello, debe hacer clic en el primer elemento, después oprimir y mantener oprimida la tecla *Mayús* mientras hace clic en el último elemento a seleccionar en el rango.
- Una lista `MULTIPLE_INTERVAL_SELECTION` permite la selección de un rango continuo, según el procedimiento descrito para una lista `SINGLE_INTERVAL_SELECTION`. Dicha lista también permite la selección de elementos variados si se mantiene oprimida la tecla *Ctrl* mientras se hace clic en cada elemento que se desea seleccionar.
- La clase `JList` no proporciona una barra de desplazamiento si hay más elementos en la lista que el número de filas visibles. Un objeto `JScrollPane` se utiliza para proporcionar la herramienta de desplazamiento.
- El método `setFixedCellHeight` de `JList` especifica la altura, en píxeles, de cada elemento en un objeto `JList`. El método `setFixedCellWidth` establece la anchura, en píxeles, de un objeto `JList`.
- Normalmente, un evento generado por otro componente de la GUI (al que se le conoce como evento externo) especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`.
- El método `setListData` de `JList` establece los elementos mostrados en un objeto `JList`. El método `getSelectedValues` devuelve los elementos seleccionados como un arreglo de objetos `Object`.
- Los eventos de ratón pueden atraparse para cualquier componente de la GUI que se derive de `java.awt.Component`, utilizando objetos `MouseListener` y `MouseMotionListener`.
- Cada método manejador de eventos de ratón toma como su argumento a un objeto `MouseEvent`, el cual contiene información acerca del evento de ratón y de la ubicación en donde ocurrió el evento.
- Los métodos `addMouseListener` y `addMouseMotionListener` son métodos de `Component` que se utilizan para registrar componentes de escucha de eventos, para un objeto de cualquier clase que extienda a `Component`.
- Muchas de las interfaces de escucha de eventos proporcionan varios métodos. Para cada uno de ellos existe su correspondiente clase adaptadora de escucha de eventos, la cual proporciona una implementación detallada de cada uno de los métodos en la interfaz. El programador puede extender la clase adaptadora para heredar la implementación predeterminada de cada método, y simplemente sobrescribir el método o métodos necesarios para el manejo de eventos en el programa.
- El método `getClickCount` de `MouseEvent` devuelve el número de clics de ratón que ocurrieron.
- Los métodos `isMetaDown` e `isAltDown` de `InputEvent` se utilizan para determinar con cuál botón del ratón hizo clic el usuario.
- Los objetos `KeyListener` manejan eventos clave que se generan cuando se oprimen y sueltan teclas en el teclado. Un objeto `KeyListener` debe proporcionar declaraciones para los métodos `keyPressed`, `keyReleased` y `keyTyped`, cada uno de los cuales recibe un objeto `KeyEvent` como argumento.
- El método `keyPressed` es llamado en respuesta a la acción de oprimir cualquier tecla. El método `keyTyped` es llamado en respuesta a la acción de oprimir una tecla que no sea una *tecla de acción* (es decir, cualquier tecla de dirección, *Inicio*, *Fin*, *Re Pág*, *Av Pág*, cualquier tecla de función, *Bloq Num*, *Impr Pant*, *Bloq Despl*, *Bloq Mayús* y *Pausa*.) El método `keyReleased` es llamado cuando la tecla se suelta después de un evento `keyPressed` o `keyTyped`.
- El método `getKeyCode` de `KeyEvent` obtiene el código de tecla virtual de la tecla que se haya oprimido. La clase `KeyEvent` mantiene un conjunto de constantes de código de tecla virtual, que representan a cada tecla en el teclado.
- El método `getKeyText` de `KeyEvent` devuelve un objeto `String` que contiene el nombre de la tecla que corresponde a su argumento de código de tecla virtual. El método `getKeyChar` obtiene el valor Unicode del carácter escrito. El método `isActionKey` determina si la tecla en el evento era una tecla de acción.
- El método `getModifiers` de `InputEvent` determina si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla. El método `getKeyModifiersText` de `KeyEvent` produce una cadena que contiene los nombres de las teclas modificadoras que se hayan oprimido.
- Los administradores de esquemas ordenan los componentes de la GUI en un contenedor, para fines de presentación.
- El administrador de esquemas `FlowLayout` distribuye los componentes de izquierda a derecha, en el orden en el que se agregan al contenedor. Cuando se llega al borde derecho del contenedor, los componentes continúan en la siguiente línea.
- El método `setAlignment` de `FlowLayout` cambia la alineación del objeto `FlowLayout` a `FlowLayout.LEFT`, `FlowLayout.CENTER` o `FlowLayout.RIGHT`.
- El administrador de esquemas `BorderLayout` ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. Puede agregarse solamente un componente a cada región.
- El método `layoutContainer` de `LayoutManager` recalcula el esquema de su argumento `Container`.
- El administrador de esquemas `GridLayout` divide el contenedor en una cuadrícula de filas y columnas. Los componentes se agregan a un objeto `GridLayout` empezando en la celda superior izquierda, y continúan agregándose de izquierda a derecha hasta que la fila está llena. Después el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente.

- El método `validate` de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI mostrados en pantalla.
- Los paneles se crean con la clase `JPanel`, la cual extiende a `JComponent`. Se pueden agregar componentes a los objetos `JPanel`, incluyendo otros componentes.

TERMINOLOGÍA

`.gif`, extensión de nombre de archivo
`.jpg`, extensión de nombre de archivo
`.png`, extensión de nombre de archivo
Abstract Windows Toolkit
`AbstractButton`, clase
`ActionEvent`, clase
`ActionListener`, interfaz
`actionPerformed`, método
`add`, método de `ButtonGroup`
`add`, método de la clase `Container`
`addItemListener`, método
`addKeyListener`, método
`addListSelectionListener`, método
`addMouseListener`, método
`addMouseMotionListener`, método
administrador de esquemas
alineado a la derecha
alineado a la izquierda
apariencia visual
apariencia visual adaptable
arrastrar
barra de desplazamiento
barra de herramientas
barra de menús
`BorderLayout`, clase
`BorderLayout.CENTER`
`BorderLayout.EAST`
`BorderLayout.NORTH`
`BorderLayout.SOUTH`
`BorderLayout.WEST`
botón
botón de comando
botón de opción
`ButtonGroup`, clase
casilla de verificación
centrado
clase adaptadora
`Component`, clase
`ComponentAdapter`, clase
componente de escucha de eventos
componente de la GUI
componente ligero
`ComponentListener`, interfaz
`Container`, clase
`ContainerAdapter`, clase
`ContainerListener`, interfaz
contraseña
control
controlado por eventos
cuadro de desplazamiento
despachar un evento
enfoque
"escuchar" un evento
espacio libre horizontal
espacio libre vertical
etiqueta
etiqueta de botón
etiqueta de casilla de verificación
`EventListenerList`, clase
evento
`EventObject`, clase
flecha de desplazamiento
`FlowLayout`, clase
`FlowLayout.CENTER`
`FlowLayout.LEFT`
`FlowLayout.RIGHT`
`FocusAdapter`, clase
`FocusListener`, interfaz
`Font.BOLD`
`Font.ITALIC`
`Font.PLAIN`
Formato de intercambio de gráficos (GIF)
`getActionCommand`, método
`getClickCount`, método
`getIcon`, método
`getKeyChar`, método de `KeyEvent`
`getKeyCode`, método de `KeyEvent`
`getKeyModifiersText`, método
`getKeyText`, método de `KeyEvent`
`getModifiers`, método de `InputEvent`
`getPassword`, método de `JPasswordField`
`getPoint`, método de `MouseEvent`
`getSelectedIndex`, método de `JComboBox`
`getSelectedIndex`, método de `JList`
`getSelectedValues`, método de `JList`
`getSource`, método de `ActionEvent`
`getStateChange`, método de `ItemEvent`
`getText`, método de `JLabel`
`getX`, método de `MouseEvent`
`getY`, método de `MouseEvent`
gráficos portables de red (PNG)
`GridLayout`, clase
Grupo unido de expertos en fotografía (JPEG)
`Icon`, interfaz
icono de sustitución
ID de evento
`ImageIcon`, clase
información sobre herramientas
`InputEvent`, clase
interfaz de escucha de eventos
`isActionKey`, método de `KeyEvent`
`isAltDown`, método de `InputEvent`
`isMetaDown`, método de `InputEvent`
`isSelected`, método de `JCheckBox`
`ItemEvent`, clase
`ItemListener`, interfaz
`itemStateChanged`, método de `ItemListener`
Java Foundation Classes
`java.awt`, paquete
`java.awt.event`, paquete
`javax.swing`, paquete
`javax.swing.event`, paquete
`JButton`, clase
`JCheckBox`, clase
`JComboBox`, clase
`JComponent`, clase

http://librosysolucionarios.net

JLabel, clase
JList, clase
JPanel, clase
JPasswordField, clase
JRadioButton, clase
JScrollPane, clase
JTextComponent, clase
JTextField, clase
JToggleButton, clase
justificado a la izquierda
KeyAdapter, clase
KeyEvent, clase
KeyListener, interfaz
keyPressed, método de KeyListener
keyReleased, método de KeyListener
keyTyped, método de KeyListener
layoutContainer, método de LayoutManager
LayoutManager, interfaz
lista de selección múltiple
lista de selección simple
lista desplegable
ListSelectionEvent, clase
ListSelectionListener, interfaz
ListSelectionModel, interfaz
localización de la interfaz de usuario
manejador de eventos
manejar un evento
menú
modelo de delegación de eventos
modo de selección
MouseAdapter, clase
mouseClicked, método de MouseListener
mouseDragged, método de MouseMotionListener
mouseEntered, método de MouseListener
MouseEvent, clase
mouseExited, método de MouseListener
MouseListener, interfaz
MouseMotionAdapter, clase
MouseMotionListener, interfaz
mouseMoved, método de MouseMotionListener
mousePressed, método de MouseListener
mouseReleased, método de MouseListener
nemónicos
Point, clase
registrar un componente de escucha de eventos
setAlignment, método de FlowLayout
setBackground, método de Component
setEditable, método de JTextComponent
setFixedCellHeight, método de JList
setFixedCellWidth, método de JList
setHorizontalAlignment, método de JLabel
setHorizontalTextPosition, método de JLabel
setIcon, método
setLayout, método de la clase Container
setListData, método de JList
setMaximumRowCount, método de JComboBox
setRolloverIcon, método de AbstractButton
setSelectionMode, método de JList
setToolTipText, método de JComponent
setVerticalAlignment, método de JLabel
setVerticalTextPosition, método de JLabel
setVisible, método de Component
setVisibleRowCount, método de JList
sistema de ventanas
SwingConstants, interfaz
tecla de método abreviado (nemónicos)
texto de sólo lectura
validate, método
valueChanged, método de ListSelectionListener
ventana
widget (accesorio de ventana)
WindowAdapter, clase
windowClosing, método de WindowListener
WindowListener, interfaz

EJERCICIOS DE AUTOEVALUACIÓN

13.1 Complete las siguientes oraciones:

a) El método _____ es llamado cuando el ratón se mueve sin oprimir los botones y un componente de escucha de eventos está registrado para manejar el evento.
b) El texto que no puede ser modificado por el usuario se llama texto _____.
c) Un _____ ordena los componentes de la GUI en un objeto `Container`.
d) El método `add` para adjuntar componentes de la GUI es un método de la clase _____.
e) GUI es un acrónimo para _____.
f) El método _____ se utiliza para especificar el administrador de esquemas para un contenedor.
g) Una llamada al método `mouseDragged` va precedida por una llamada al método _____ y va seguida de una llamada al método _____.

13.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) `BorderLayout` es el administrador de esquemas predeterminado para un panel de contenido.
b) Cuando el cursor del ratón se mueve hacia los límites de un componente de la GUI, se hace una llamada al método `mouseOver`.
c) Un objeto `JPanel` no puede agregarse a otro `JPanel`.
d) En un esquema `BorderLayout`, dos botones que se agreguen a la región `NORTH` se mostrarán uno al lado del otro.
e) Cuando se utiliza `BorderLayout`, sólo deben mostrarse un máximo de cinco componentes.

13.3 Encuentre el (los) error(es) en cada una de las siguientes instrucciones y explique cómo corregirlo(s).

a) `nombreBoton = JButton( "Leyenda" );`
b) `JLabel unaEtiqueta, JLabel; // crear referencias`
c) `campoTexto = new JTextField( 50, "Texto predeterminado" );`
d)
```
Container contenedor = getContentPane();
setLayout( new BorderLayout() );
boton1 = new JButton( "Estrella del norte" );
boton2 = new JButton( "Polo sur" );
contenedor.add( boton1 );
contenedor.add( boton2 );
```

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

13.1 a) `mouseMoved`. b) No editable (de sólo lectura). c) Administrador de esquemas. d) `Container`. e) interfaz gráfica de usuario. f) `setLayout`. g) `mousePressed`, `mouseReleased`.

13.2 a) Verdadero.
b) Falso. Se hace una llamada al método `mouseEntered`.
c) Falso. Un `JPanel` puede agregarse a otro `JPanel`, ya que `JPanel` es una subclase indirecta de `Component`. Por lo tanto, un `JPanel` es un `Component`. Cualquier `Component` puede agregarse a un `Container`.
d) Falso. Sólo se mostrará el último botón que se agregue. Recuerde que sólo debe agregarse un componente a cada región en un esquema `BorderLayout`.
e) Verdadero.

13.3 a) Se necesita `new` para crear un objeto.
b) `JLabel` es el nombre de una clase y no puede utilizarse como nombre de variable.
c) Los argumentos que se pasan al constructor están invertidos. El objeto `String` debe pasarse primero.
d) Se ha establecido `BorderLayout` y los componentes se agregarán sin especificar la región, por lo que ambos se agregarán a la región central. Las instrucciones `add` apropiadas serían:

```
contenedor.add( boton1, BorderLayout.NORTH );
contenedor.add( boton2, BorderLayout.SOUTH );
```

EJERCICIOS

13.4 Complete las siguientes oraciones:

a) La clase `JTextField` extiende directamente a la clase _____.
b) Enliste tres administradores de esquemas de este capítulo: _____, _____ y _____.
c) El método _____ de `Container` adjunta un componente de la GUI a un contenedor.
d) El método _____ es llamado cuando se suelta uno de los botones del ratón (sin mover el ratón).
e) La clase _____ se utiliza para crear un grupo de objetos `JRadioButton`.

13.5 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Sólo puede usarse un administrador de esquemas por cada objeto `Container`.
b) Los componentes de la GUI pueden agregarse a un objeto `Container` en cualquier orden, en un esquema `BorderLayout`.
c) Los objetos `JRadioButton` proporcionan una serie de opciones mutuamente exclusivas (es decir, sólo uno puede ser `true` en un momento dado).
d) El método `setFont` de `Graphics` se utiliza para establecer el tipo de letra para los campos de texto.
e) Un objeto `JList` muestra una barra de desplazamiento si hay más elementos en la lista de los que puedan mostrarse en pantalla.
f) Un objeto `Mouse` tiene un método llamado `mouseDragged`.

13.6 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Un objeto `JApplet` no tiene panel de contenido.
b) Un objeto `JPanel` es un objeto `JComponent`.
c) Un objeto `JPanel` es un objeto `Component`.
d) Un objeto `JLabel` es un objeto `Container`.
e) Un objeto `JList` es un objeto `JPanel`.

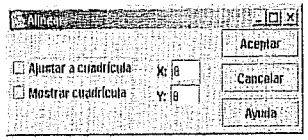
http://librosysolucionarios.net

f) Un objeto `AbstractButton` es un objeto `JButton`.
g) Un objeto `JTextField` es un objeto `Object`.
h) `ButtonGroup` es una subclase de `JComponent`.

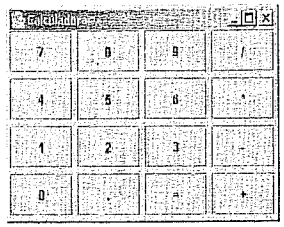
13.7 Encuentre los errores en cada una de las siguientes líneas de código y explique cómo corregirlos.

a) `import javax.swing.*  // incluir el paquete swing`
b) `objetoPanel.GridLayout( 8, 8 );  // establecer esquema GridLayout`
c) `contenedor.setLayout( new FlowLayout( FlowLayout.DEFAULT ) );`
d) `contenedor.add( botonEste, EAST );  // BorderLayout`

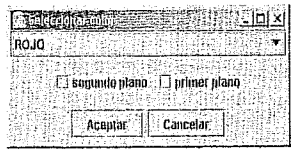
13.8 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



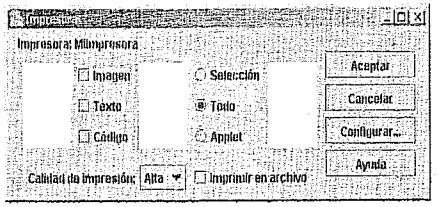
13.9 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



13.10 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



13.11 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



13.12 Escriba un programa de conversión de temperatura, que convierta de grados Fahrenheit a Centígrados. La temperatura en grados Fahrenheit deberá introducirse desde el teclado (mediante un objeto `JTextField`). Debe usarse un objeto `JLabel` para mostrar la temperatura convertida. Use la siguiente fórmula para la conversión:

$$Centígrados = \frac{5}{9} \times (Fahrenheit - 32)$$

13.13 Mejore el programa de conversión de temperatura del ejercicio 13.12, agregando la escala de temperatura Kelvin. Además, el programa debe permitir al usuario realizar conversiones entre las dos escalas. Use la siguiente fórmula para la conversión entre Kelvin y Centígrados (además de la fórmula del ejercicio 13.12):

Kelvin = Centígrados + `273.15`

13.14 Escriba una aplicación que permita al usuario dibujar un rectángulo, arrastrando el ratón en la ventana de la aplicación. La coordenada superior izquierda deberá ser la ubicación en donde el usuario oprima el botón del ratón, y la coordenada inferior derecha deberá ser la ubicación en donde el usuario suelte el botón del ratón. Además, muestre el área del rectángulo en un objeto `JLabel`, en la región `SOUTH` de un esquema `BorderLayout`. Use la siguiente fórmula para el área:

área = anchura × altura

13.15 Modifique el programa del ejercicio 13.14 para dibujar distintas figuras. El usuario deberá poder elegir de entre un óvalo, un arco, una línea y un rectángulo con esquinas redondeadas. Además, muestre las coordenadas del ratón en la barra de estado.

13.16 Escriba un programa que permita al usuario dibujar una figura con el ratón. La figura a dibujar deberá determinarse mediante un evento `KeyEvent`, utilizando las siguientes teclas: *c* dibuja un círculo, *o* dibuja un óvalo, *r* dibuja un rectángulo y *l* dibuja una línea. El tamaño y la ubicación de la figura deberán determinarse mediante los eventos `mousePressed` y `mouseReleased`. Muestre el nombre de la figura actual en un objeto `JLabel` que se encuentre en la región `SOUTH` de un esquema `BorderLayout`. La figura inicial predeterminada deberá ser un círculo.

13.17 Cree una aplicación que permita al usuario pintar una imagen. El usuario deberá ser capaz de seleccionar la figura a dibujar, el color en el que debe aparecer y si debe rellenarse con color. Use los componentes de interfaz gráfica de usuario que vimos en este capítulo, como `JComboBox`, `JRadioButton` y `JCheckBox`, para permitir al usuario seleccionar varias opciones. El programa deberá proporcionar un objeto `JButton` que permita al usuario borrar el contenido de la ventana.

13.18 Escriba un programa que utilice instrucciones `System.out.println` para imprimir los eventos según vayan ocurriendo. Proporcione un objeto `JComboBox` con un mínimo de cuatro elementos. El usuario deberá ser capaz de seleccionar del objeto `JComboBox` un evento a vigilar. Cuando ocurra ese evento específico, muestre información acerca del mismo en un cuadro de diálogo de mensaje. Use el método `toString` en el objeto evento para convertirlo en una representación de cadena.

13.19 Escriba un programa que dibuje un cuadrado. A medida que el ratón se desplace sobre el área de dibujo, vuelva a pintar el cuadrado de manera que la esquina superior izquierda de éste siga la ruta exacta del cursor del ratón.

13.20 Modifique el programa de la figura 13.19 para incorporar colores. Permita al usuario seleccionar el color con objetos `JRadioButton` que representen los siguientes seis colores: rojo, negro, magenta, azul, verde y amarillo. Cuando se seleccione un nuevo color, la acción de dibujar deberá realizarse en el nuevo color.

13.21 Escriba un programa que juegue a "adivinar el número" de la siguiente manera: su programa debe elegir el número a adivinar, seleccionando un entero al azar en el rango de 1 a 1000. El programa entonces deberá mostrar lo siguiente en una etiqueta:

```
Tengo un número entre 1 y 1000. ¿Puede usted adivinarlo?
Por favor escriba su primer intento.
```

Debe usarse un objeto `JTextField` para introducir el intento. A medida que se introduzca cada intento, el color de fondo deberá cambiar ya sea a rojo o azul. Rojo indica que el usuario se está "acercando" y azul indica que el usuario se está "alejando". Un objeto `JLabel` deberá mostrar el mensaje `"Demasiado alto"` o `"Demasiado bajo"` para ayudar al usuario a tratar de adivinar correctamente el número. Cuando el usuario adivine correctamente, deberá mostrarse el mensaje `"¡Correcto!"`, y el objeto `JTextField` utilizado para la entrada deberá cambiar para que no pueda editarse. Debe proporcionarse un objeto `JButton` para permitir al usuario jugar de nuevo. Cuando se haga clic en el objeto `JButton`, deberá generarse un nuevo número aleatorio y el objeto `JTextField` de entrada deberá cambiar para poder editarse otra vez.

13.22 A menudo es conveniente mostrar los eventos que ocurren durante la ejecución de un programa. Esto puede ayudarle a comprender cuándo ocurren los eventos y cómo se generan. Escriba un programa que permita al usuario generar y procesar cada uno de los eventos descritos en este capítulo. El programa deberá proporcionar métodos de las interfaces `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` y `KeyListener`, para mostrar mensajes cuando ocurran los eventos. Use el método `toString` para convertir los objetos evento que se reciban en cada manejador de eventos, en un objeto `String` que pueda mostrarse. El método `toString` crea un objeto `String` que contiene toda la información del objeto evento.

13.23 Modifique su solución al ejercicio 13.17 para permitir al usuario seleccionar un tipo de letra y un tamaño de tipo de letra, y también que pueda escribir en un objeto `JTextField`. Cuando el usuario oprima *Intro*, el texto deberá mostrarse en el tamaño y tipo de letra seleccionados. Modifique aún más el programa para permitir al usuario especificar la posición exacta en la que debe mostrarse el texto.

http://librosysolucionarios.net

13.24 Escriba un programa que permita al usuario seleccionar una figura de un objeto `JComboBox` y que dibuje esa figura 20 veces, con ubicaciones y dimensiones aleatorias en el método `paint`. El primer elemento en el objeto `JComboBox` deberá ser la figura predeterminada que se muestre la primera vez que se llame a `paint`.

13.25 Modifique el ejercicio 13.24 para dibujar cada una de las 20 figuras de tamaño aleatorio en un color seleccionado al azar. Use los 13 objetos `Color` predefinidos en un arreglo de objetos `Color`.

13.26 Modifique el ejercicio 13.25 para permitir al usuario seleccionar el color en el que deben dibujarse las figuras, mediante un cuadro de diálogo `JColorChooser`.

13.27 Escriba un programa utilizando métodos de la interfaz `MouseListener`, el cual permita al usuario oprimir el botón del ratón, arrastrar el ratón y soltar el botón del ratón. Cuando se suelte el botón del ratón, dibuje un rectángulo con la esquina superior izquierda, anchura y altura apropiadas. (*Sugerencia:* El método `mousePressed` deberá capturar el conjunto de coordenadas en donde el usuario oprima y mantenga oprimido el botón del ratón inicialmente, y el método `mouseReleased` deberá capturar el conjunto de coordenadas en donde el usuario suelte el botón del ratón. Ambos métodos deberán almacenar los valores de coordenada apropiados. Todos los cálculos de la anchura, altura y esquina superior izquierda deberán realizarse mediante el método `paint`, antes de que se dibuje la figura.)

13.28 Modifique el ejercicio 13.27 para proporcionar un efecto de "banda de hule". A medida que el usuario arrastre el ratón, deberá poder ver el tamaño actual del rectángulo para saber exactamente cómo se verá el rectángulo cuando se suelte el botón del ratón. (*Sugerencia:* El método `mouseDragged` deberá realizar las mismas tareas que `mouseReleased`.)

13.29 Modifique el ejercicio 13.28 para permitir al usuario seleccionar cuál figura dibujar. Un objeto `JComboBox` deberá proporcionar opciones que incluyan, cuando menos, rectángulo, óvalo, línea y rectángulo redondeado.

13.30 Modifique el ejercicio 13.29 para permitir al usuario seleccionar el color de dibujo de un cuadro de diálogo `JColorChooser`.

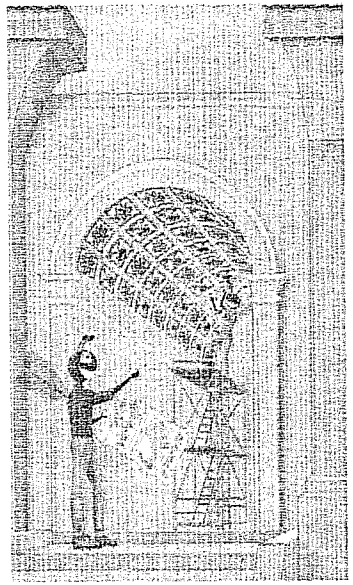
13.31 Modifique el ejercicio 13.30 para permitir al usuario especificar si una figura debe llenarse o vaciarse cuando se dibuje. El usuario deberá hacer clic en un objeto `JCheckBox` para indicar llena o vacía.

14

Componentes de la interfaz gráfica de usuario: Parte 2

Objetivos

- Crear y manipular áreas de texto, controles deslizables, menús, menús contextuales y ventanas.
- Crear objetos `JPanel` personalizados.
- Cambiar la apariencia visual de una GUI, utilizando la apariencia visual adaptable (PLAF) de Swing.
- Crear una interfaz de múltiples documentos con `JDesktopPane` y `JInternalFrame`.
- Utilizar los administradores de esquemas adicionales.



Yo no aclamo haber controlado los eventos, sino que confieso plenamente que los eventos me han controlado a mí.
Abraham Lincoln

Un buen símbolo es el mejor argumento, y es misionero para persuadir a miles.
Ralph Waldo Emerson

¡Captura su realidad en pintura!
Paul Cézanne

http://librosysolucionarios.net

Plan general

14.1 Introducción
14.2 JTextArea
14.3 Creación de una subclase personalizada de JPanel
14.4 Subclase de JPanel que maneja sus propios eventos
14.5 JSlider
14.6 Ventanas: Observaciones adicionales
14.7 Uso de menús con marcos
14.8 JPopupMenu
14.9 Apariencia visual adaptable
14.10 JDesktopPane y JInternalFrame
14.11 JTabbedPane
14.12 Administradores de esquemas: BoxLayout y GridBagLayout
14.13 (Ejemplo práctico opcional) Acerca de los objetos: Modelo-vista-controlador
14.14 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes java.awt y javax.swing
14.14.1 Patrones de diseño de creación
14.14.2 Patrones de diseño de estructura
14.14.3 Patrones de diseño de comportamiento
14.14.4 Conclusión
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

14.1 Introducción

En este capítulo continuaremos nuestro estudio acerca de las GUIs. Hablaremos sobre componentes y administradores de esquemas adicionales que establecen las bases para crear las GUIs más complejas.

Empezaremos nuestra discusión con otro componente de la GUI basado en texto: `JTextArea`, el cual permite que se muestren o introduzcan varias líneas de texto. Continuaremos con ejemplos de *personalización de la clase* `JPanel`, en donde veremos cuestiones relacionadas con el proceso de pintar en componentes de la GUI de Swing. Un aspecto importante de cualquier GUI completa es un sistema de *menús* que permita al usuario realizar con efectividad tareas en el programa, por lo que demostraremos cómo crear y utilizar menús. La apariencia visual de una GUI de Swing puede ser uniforme a través de todas las plataformas en las que se ejecuta un programa de Java, o la GUI puede personalizarse mediante el uso de la *apariencia visual adaptable (PLAF)* de Swing. Proporcionaremos un ejemplo que ilustra cómo cambiar entre la *apariencia visual metálica* de Swing, una apariencia visual que simula a *Motif* (una apariencia visual de UNIX muy popular) y una que simula la apariencia visual de Microsoft Windows. Muchas de las aplicaciones de hoy en día utilizan una *interfaz de múltiples documentos (MDI)*, es decir, una ventana principal (conocida también como la *ventana padre*) que contiene a otras ventanas (a menudo conocidas como *ventanas hijas*) para administrar varios *documentos* abiertos en paralelo. Por ejemplo, muchos programas de correo electrónico le permiten tener varias ventanas abiertas al mismo tiempo, para que pueda componer o leer varios mensajes de correo electrónico. En este capítulo demostraremos el uso de las clases de Swing para crear interfaces con múltiples documentos. Finalmente, terminaremos el capítulo con una serie de ejemplos acerca de los administradores de esquemas adicionales que están disponibles para organizar interfaces gráficas de usuario.

Swing es un tema extenso y complejo. Existen muchos más componentes de la GUI y herramientas de las que podemos presentar aquí. Hablaremos sobre más componentes de la GUI de Swing en los capítulos restantes de este libro, a medida que se vayan necesitando. En nuestro libro *Advanced Java 2 Platform How to Program* hablamos sobre otros componentes y herramientas más avanzadas de Swing.

14.2 JTextArea

Un objeto `JTextArea` proporciona un área para manipular varias líneas de texto. Al igual que la clase `JTextField`, la clase `JTextArea` hereda de `JTextComponent`, la cual declara métodos comunes para objetos `JTextField`, `JTextArea` y varios otros componentes de la GUI basados en texto.

La aplicación de la figura 14.1 demuestra el uso de objetos `JTextArea`. Uno de esos objetos muestra el texto que el usuario puede seleccionar. El otro objeto `JTextArea` no puede editarse y se utiliza para mostrar el texto que el usuario seleccionó en el primer objeto `JTextArea`. Los objetos `JTextArea` no tienen eventos de acción como los objetos `JTextField`. Al igual que con los objetos `JList` de selección múltiple (sección 13.11), un evento externo de otro componente de la GUI es el que indica cuándo procesar el texto en un objeto `JTextArea`. Por ejemplo, al escribir un mensaje de correo electrónico usted normalmente hace clic en un botón **Enviar** para enviar el texto del mensaje al destinatario. De manera similar, al editar un documento en un procesador de palabras, normalmente guarda el archivo seleccionando un elemento de menú llamado **Guardar** o **Guardar como...** En este programa, el botón **Copiar >>>** genera el evento externo que copia el texto seleccionado en el objeto `JTextArea` de la izquierda, y lo muestra en el objeto `JTextArea` de la derecha.

En la línea 16, del constructor (líneas 12 a 55), se crea un *contenedor* `Box` (paquete `javax.swing`) para organizar los componentes de la GUI. La clase `Box` es una subclase de `Container` que utiliza un administrador de esquemas `BoxLayout` para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical.[1] La clase `Box` proporciona el método estático `createHorizontalBox` para crear un objeto `Box` que acomode los componentes de izquierda a derecha, en el orden en que éstos se adjunten.

```
1  // Fig. 14.1: DemoAreaTexto.java
2  // Cómo copiar texto seleccionado de un área de texto a otra.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoAreaTexto extends JFrame {
8     private JTextArea areaTexto1, areaTexto2;
9     private JButton botonCopiar;
10
11    // configurar la GUI
12    public DemoAreaTexto()
13    {
14       super( "Demostración de JTextArea" );
15
16       Box cuadro = Box.createHorizontalBox();
17
18       String cadena = "Ésta es una cadena de\ndemostración para\n" +
19          "ilustrar cómo copiar texto\nde un área de texto a \n" +
20          "otra, utilizando un\nevento externo\n";
21
22       // establecer areaTexto1
23       areaTexto1 = new JTextArea( cadena, 10, 15 );
24       cuadro.add( new JScrollPane( areaTexto1 ) );
25
26       // establecer botonCopiar
27       botonCopiar = new JButton( "Copiar >>>" );
28       cuadro.add( botonCopiar );
29       botonCopiar.addActionListener(
```

Figura 14.1 Cómo copiar texto seleccionado de un área de texto a otra. (Parte 1 de 2.)

1. En la sección 14.12 hablaremos detalladamente sobre `BoxLayout`.

http://librosysolucionarios.net

```
30
31          new ActionListener() {  // clase interna anónima
32
33             // establecer en areaTexto2 el texto seleccionado de areaTexto1
34             public void actionPerformed( ActionEvent evento )
35             {
36                areaTexto2.setText( areaTexto1.getSelectedText() );
37             }
38
39          } // fin de la clase interna anónima
40
41       ); // fin de la llamada a addActionListener
42
43       // establecer areaTexto2
44       areaTexto2 = new JTextArea( 10, 15 );
45       areaTexto2.setEditable( false );
46       cuadro.add( new JScrollPane( areaTexto2 ) );
47
48       // agregar cuadro al panel de contenido
49       Container contenedor = getContentPane();
50       contenedor.add( cuadro );    // colocar en BorderLayout.CENTER
51
52       setSize( 425, 200 );
53       setVisible( true );
54
55    } // fin del constructor de DemoAreaTexto
56
57    public static void main( String args[] )
58    {
59       DemoAreaTexto aplicacion = new DemoAreaTexto();
60       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
61    }
62
63 } // fin de la clase DemoAreaTexto
```

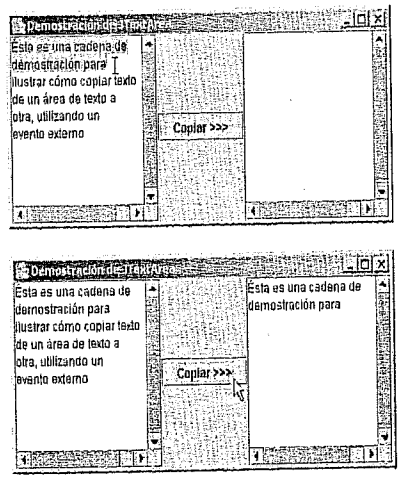


Figura 14.1 Cómo copiar texto seleccionado de un área de texto a otra. (Parte 2 de 2.)

http://librosysolucionarios.net

La aplicación crea los objetos `JTextArea` llamados `areaTexto1` (línea 23) y `areaTexto2` (línea 44). Cada objeto `JTextArea` tiene 10 filas y 15 columnas. En la línea 23 se especifica que `cadena` debe mostrarse como el contenido predeterminado del objeto `JTextArea`. Un objeto `JTextArea` no proporciona barras de desplazamiento si no puede mostrar su contenido completo. Por esta razón, en la línea 24 se crea un objeto `JScrollPane`, se inicializa con `areaTexto1` y se adjunta al contenedor `cuadro`. De manera predeterminada aparecerán las barras de desplazamiento horizontal y vertical, según sea necesario.

En las líneas 27 a 41 se crea el objeto `JButton` llamado `botonCopiar` con la etiqueta "`Copiar >>>`," se agrega `botonCopiar` al contenedor `cuadro` y se registra el manejador de eventos para los eventos `ActionEvent` de `botonCopiar`. Este botón proporciona el evento externo que determina el momento en que el programa debe copiar el texto seleccionado en `areaTexto1` a `areaTexto2`. Cuando el usuario hace clic en `botonCopiar`, la línea 36 en `actionPerformed` indica que el método *`getSelectedText`* (heredado en `JTextArea` de `JTextComponent`) debe devolver el *texto seleccionado* de `areaTexto1`. El usuario selecciona texto arrastrando el ratón sobre el texto deseado para resaltarlo. El método `setText` cambia el texto en `areaTexto2` a la cadena devuelta por `getSelectedText`.

En las líneas 44 a 46 se crea `areaTexto2` y se agrega al contenedor `cuadro`. En las líneas 49 y 50 se obtiene el panel de contenido para la ventana y se agrega `cuadro` a este panel de contenido. Recuerde que en la sección 13.15 vimos que el esquema predeterminado de un panel de contenido es `BorderLayout`, y que el método `add` adjunta de manera predeterminada su argumento a `CENTER` del esquema `BorderLayout`.

En ocasiones es conveniente hacer que, cuando el texto llega al lado derecho de un objeto `JTextArea`, pase a la siguiente línea. A esto se le conoce como *envoltura de línea*. De manera predeterminada, un objeto `JTextArea` no envuelve líneas.

Observación de apariencia visual 14.1

Para proporcionar la funcionalidad de envoltura de líneas a un objeto `JTextArea`, debe invocar al método `setLineWrap` de `JTextArea` con un argumento `true`.

Políticas de barra de desplazamiento de `JScrollPane`

En este ejemplo utilizamos un objeto `JScrollPane` para proporcionar herramientas de desplazamiento a un objeto `JTextArea`. De manera predeterminada, `JScrollPane` muestra barras de desplazamiento solamente si se requieren. Usted puede establecer las *políticas de barra de desplazamiento* horizontal y vertical para el objeto `JScrollPane` al momento de su creación. Si un programa tiene una referencia a un objeto `JScrollPane`, el programa puede utilizar los métodos *`setHorizontalScrollBarPolicy`* y *`setVerticalScrollBarPolicy`* de `JScrollPane` para cambiar las políticas de barra de desplazamiento en cualquier momento. La clase `JScrollPane` declara las constantes:

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que siempre debe aparecer la barra de desplazamiento, las constantes:

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que debe aparecer la barra de desplazamiento sólo si es necesario (los valores predeterminados), y las constantes:

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que nunca debe aparecer una barra de desplazamiento. Si la política de barra de desplazamiento horizontal se establece en `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, entonces un objeto `JTextArea` que se adjunte al objeto `JScrollPane` envolverá las líneas.

14.3 Creación de una subclase personalizada de `JPanel`

En el capítulo 13 demostramos que los objetos `JPanel` pueden agregar un conjunto de componentes de la GUI, para distribuirlos mejor en la pantalla. Los objetos `JPanel` son bastante flexibles. Otros de los usos para los objetos `JPanel` son para crear *áreas dedicadas de dibujo* y crear áreas que reciban eventos de ratón. En ocasiones, los programas extienden a la clase `JPanel` para crear nuevos componentes de la GUI. En nuestro siguiente ejemplo utilizamos un objeto `JPanel` para crear un área dedicada de dibujo, para separar el dibujo del resto de la interfaz gráfica de usuario del programa. Esto puede ser beneficioso en las interfaces gráficas de usuario de Swing. Si los gráficos y los componentes de la GUI de Swing no se muestran en el orden correcto, es posible que los componentes de la GUI no se muestren correctamente.

Observación de apariencia visual 14.2

Al combinar gráficos y componentes de la GUI de Swing, tal vez puedan llegar a mostrarse incorrectamente los gráficos, los componentes de la GUI o ambos. Al utilizar objetos `JPanel` para dibujar se puede eliminar este problema, ya que se proporciona un área dedicada para gráficos.

Como pintar en una subclase de `JComponent` con el método `paintComponent`

Los componentes de Swing que heredan de la clase `JComponent` contienen el método `paintComponent` para dibujar en el contexto de una GUI ligera de Swing. Al personalizar un objeto `JPanel` para usarlo como un área dedicada de dibujo, la subclase debe sobrescribir al método `paintComponent`, y debe hacer una llamada a la versión de `paintComponent` de la superclase; ésta debe ser la primera instrucción en el cuerpo del método sobrescrito. Esto asegura que la acción de pintar ocurra en el orden apropiado, y que el mecanismo de Swing para pintar lo haga correctamente. Por ejemplo, una parte importante de este mecanismo es que las subclases de `JComponent` deben soportar la *transparencia*, que puede establecerse con el método `setOpaque` (un argumento `false` indica que el componente es transparente). Para pintar un componente correctamente, el programa debe determinar si este componente es transparente. El código para esto se encuentra en la implementación del método `paintComponent` que pertenece a la superclase. Cuando un componente es transparente, `paintComponent` no borra el fondo del componente cuando el programa pinta a este componente. Cuando un componente es *opaco*, `paintComponent` borra el fondo antes de continuar con la operación de pintar. Si no se llama a la versión de `paintComponent` correspondiente a la superclase, un componente de la GUI opaco generalmente no se mostrará de manera correcta en la interfaz de usuario. Además, si la versión de la superclase es llamada antes de ejecutar las instrucciones de dibujo personalizadas, los resultados generalmente se borrarán.

Observación de apariencia visual 14.3

Al sobrescribir el método `paintComponent` de un objeto `JComponent`, la primera instrucción en el cuerpo debe ser siempre una llamada al método `paintComponent` de la superclase.

Error común de programación 14.1

Al sobrescribir el método `paintComponent` de `JComponent`, si no se llama a la versión original de `paintComponent`, que se encuentra en la superclase, el componente podría mostrarse incorrectamente.

Error común de programación 14.2

Al sobrescribir el método `paintComponent` de un objeto `JComponent`, si se llama al método `paintComponent` de la superclase después de realizar otros dibujos, éstos se borrarán.

Las clases `JFrame` y `JApplet` no son subclases de `JComponent`; por lo tanto, no contienen el método `paintComponent`. Para dibujar directamente en subclases de `JFrame` y `JApplet`, debe sobrescribir el método `paint`.

Observación de apariencia visual 14.4

Al llamar al método `repaint` para un componente de la GUI de Swing, se indica que ese componente deberá pintarse lo más pronto posible. El fondo del componente de la GUI se borra solamente si este componente es opaco. La mayoría de los componentes de Swing son transparentes de manera predeterminada. El método `setOpaque` de `JComponent` puede recibir un argumento `boolean` que indique si el componente es opaco (`true`) o transparente (`false`). Los componentes de la GUI de AWT son distintos de los componentes de Swing en cuanto a que `repaint` resulta en una llamada al método `update` de `Component` (el cual borra el fondo del componente), y `update` llama al método `paint` (en vez de llamar a `paintComponent`).

http://librosysolucionarios.net

Definición del área personalizada de dibujo

El programa de las figuras 14.2 y 14.3 demuestra el uso de una subclase personalizada de `JPanel`. La clase `PanelPersonalizado` (figura 14.2) tiene su propio método `paintComponent` que dibuja un círculo o un cuadrado, dependiendo del valor que se pase al método `dibujar` de `PanelPersonalizado`. Para este fin, en la línea 7 se declaran constantes que permiten al programa especificar la figura que un `PanelPersonalizado` dibuja en sí mismo con cada llamada a su método `paintComponent`. La clase `PruebaPanelPersonalizado` (figura 14.3) crea un objeto `PanelPersonalizado` y una GUI que permiten al usuario seleccionar la figura a dibujar.

La clase `PanelPersonalizado` contiene un campo, `figura` (línea 8 de la figura 14.2), que almacena un entero que representa la figura a dibujar. El método `paintComponent` (líneas 11 a 19) dibuja una figura en el panel. Si `figura` es `CIRCULO`, el método `fillOval` de `Graphics` dibuja un círculo relleno. Si `figura` es `CUADRADO`, el método `fillRect` de `Graphics` dibuja un cuadrado relleno. El método `dibujar` (líneas 22 a 26) establece el campo `figura` y llama a `repaint` para actualizar el objeto `PanelPersonalizado`. Observe que al llamar a `repaint` para el objeto `PanelPersonalizado` se programa una operación de pintura sólo para el objeto `PanelPersonalizado`, no para toda la GUI. El método `paintComponent` será llamado para pintar nuevamente el objeto `PanelPersonalizado` y dibujar la figura apropiada.

El constructor de `PruebaPanelPersonalizado` (figura 14.3, líneas 13 a 66) crea un objeto `PanelPersonalizado` (línea 18) y establece su color de fondo en verde (línea 19), por lo que el área del `PanelPersonalizado` es visible en la aplicación. Todos los objetos `JPanel` (`PanelPersonalizado` extiende a `JPanel`) son opacos de manera predeterminada, por lo que se muestran sus colores de fondo. El constructor crea los objetos `JButton` llamados `botonCuadrado` (línea 22) y `botonCirculo` (línea 37). En las líneas 23 a 35 se registra el manejador de eventos de `botonCuadrado`. En las líneas 38 a 50 se registra el manejador de eventos de `botonCirculo`. Cada uno de los métodos `actionPerformed` llama al

```
1  // Fig. 14.2: PanelPersonalizado.java
2  // Una clase JPanel personalizada.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class PanelPersonalizado extends JPanel {
7     public final static int CIRCULO = 1, CUADRADO = 2;
8     private int figura;
9
10    // usar figura para dibujar un óvalo o rectángulo
11    public void paintComponent( Graphics g )
12    {
13       super.paintComponent( g );
14
15       if ( figura == CIRCULO )
16          g.fillOval( 50, 10, 60, 60 );
17       else if ( figura == CUADRADO )
18          g.fillRect( 50, 10, 60, 60 );
19    }
20
21    // establecer valor de figura y repintar PanelPersonalizado
22    public void dibujar( int figuraADibujar )
23    {
24       figura = figuraADibujar;
25       repaint();
26    }
27
28 } // fin de la clase PanelPersonalizado
```

Figura 14.2 Subclase de `JPanel` utilizada como un área personalizada de dibujo.

método `dibujar` de `PanelPersonalizado` (líneas 30 y 45). En cada caso, se pasa la constante apropiada (`PanelPersonalizado.CUADRADO` o `PanelPersonalizado.CIRCULO`) como argumento para indicar qué figura debe dibujar el objeto `PanelPersonalizado`.

```
1  // Fig. 14.3: PruebaPanelPersonalizado.java
2  // Uso de un objeto JPanel personalizado.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaPanelPersonalizado extends JFrame {
8     private JPanel panelBotones;
9     private PanelPersonalizado miPanel;
10    private JButton botonCirculo, botonCuadrado;
11
12    // configurar la GUI
13    public PruebaPanelPersonalizado()
14    {
15       super( "Prueba de PanelPersonalizado" );
16
17       // crear área personalizada de dibujo
18       miPanel = new PanelPersonalizado();
19       miPanel.setBackground( Color.GREEN );
20
21       // establecer botonCuadrado
22       botonCuadrado = new JButton( "Cuadrado" );
23       botonCuadrado.addActionListener(
24
25          new ActionListener() {  // clase interna anónima
26
27             // dibujar un cuadrado
28             public void actionPerformed( ActionEvent evento )
29             {
30                miPanel.dibujar( PanelPersonalizado.CUADRADO );
31             }
32
33          } // fin de la clase interna anónima
34
35       ); // fin de la llamada a addActionListener
36
37       botonCirculo = new JButton( "Círculo" );
38       botonCirculo.addActionListener(
39
40          new ActionListener() {  // clase interna anónima
41
42             // dibujar un círculo
43             public void actionPerformed( ActionEvent evento )
44             {
45                miPanel.dibujar( PanelPersonalizado.CIRCULO );
46             }
47
48          } // fin de la clase interna anónima
49
50       ); // fin de la llamada a addActionListener
```

Figura 14.3 Dibujo en una subclase personalizada de `JPanel`. (Parte 1 de 2.)

http://librosysolucionarios.net


```
51
52       // establecer panel con botones
53       panelBotones = new JPanel();
54       panelBotones.setLayout( new GridLayout( 1, 2 ) );
55       panelBotones.add( botonCirculo );
56       panelBotones.add( botonCuadrado );
57
58       // adjuntar panel de botones y área personalizada de dibujo al panel de contenido
59       Container contenedor = getContentPane();
60       contenedor.add( miPanel, BorderLayout.CENTER );
61       contenedor.add( panelBotones, BorderLayout.SOUTH );
62
63       setSize( 300, 150 );
64       setVisible( true );
65
66    } // fin del constructor PruebaPanelPersonalizado
67
68    public static void main( String args[] )
69    {
70       PruebaPanelPersonalizado aplicacion = new PruebaPanelPersonalizado();
71       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
72    }
73
74 } // fin de la clase PruebaPanelPersonalizado
```

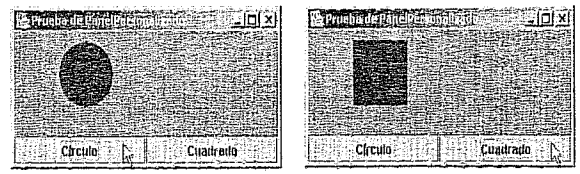


Figura 14.3 Dibujo en una subclase personalizada de `JPanel`. (Parte 2 de 2.)

Para la distribución de los botones, `PruebaPanelPersonalizado` crea un objeto `JPanel` llamado `panelBotones` (línea 53) con un esquema `GridLayout` de una fila y dos columnas (línea 54), y adjunta los botones al panel (líneas 55 y 56). Finalmente, `PruebaPanelPersonalizado` agrega `miPanel` a la región `CENTER` del panel de contenido (línea 60) y agrega `panelBotones` a la región `SOUTH` del panel de contenido (línea 61). Observe que el esquema `BorderLayout` expande a `miPanel` para llenar la región central. Ajuste el tamaño de la ventana para que vea cómo se administran la posición y el tamaño de los componentes de la GUI mediante el esquema `GridLayout` de `panelBotones` y el esquema `BorderLayout` de `PruebaPanelPersonalizado`.

14.4 Subclase de `JPanel` que maneja sus propios eventos

Los objetos `JPanel` no soportan los eventos convencionales que soportan los demás componentes de la GUI, como los botones, campos de texto y ventanas. No obstante, los objetos `JPanel` son capaces de reconocer eventos de bajo nivel, como los eventos de ratón y de teclas. El programa de las figuras 14.4 y 14.5 permite al usuario dibujar un óvalo mediante la acción de arrastrar el ratón a través de un panel. La clase `PanelAutoContenido` (figura 14.4) escucha sus propios eventos de ratón y dibuja un óvalo sobre sí misma, en respuesta a esos eventos de ratón. La ubicación y el tamaño del óvalo se determinan en base a las coordenadas de los eventos de ratón. Las coordenadas en las que el usuario oprime el ratón especifican el punto inicial para el cuadro delimitador del óvalo. A medida que el usuario arrastra el ratón, las coordenadas del puntero del ratón especifican otro punto. En conjunto, el programa utiliza estos puntos para calcular la coordenada *x*-*y* superior

izquierda, la anchura y la altura del cuadro delimitador del óvalo. El tamaño del óvalo cambia continuamente, mientras el usuario arrastra el ratón. Cuando el usuario suelta el botón del ratón, el programa calcula el cuadro delimitador final para el óvalo, y lo dibuja. En la línea 3 de la figura 14.4 se indica que la clase `PanelAutoContenido` se encuentra en el paquete `com.deitel.cpej5.cap14`, el cual permite que esta clase se importe en otros programas. La clase `PruebaPanelAutoContenido` importa a `PanelAutoContenido` en la línea 8 de la figura 14.5. Vea en la sección 8.13 la información acerca de cómo compilar y utilizar clases empaquetadas.

La clase `PruebaPanelAutoContenido` (figura 14.4) extiende a la clase `JPanel` (línea 9). Las variables de instancia `x1` y `y1` almacenan las coordenadas iniciales en donde ocurre el evento `mousePressed` en el objeto `PanelAutoContenido`. Las variables de instancia `x2` y `y2` almacenan las coordenadas en donde el usuario arrastra el ratón o suelta el botón del ratón. Todas las coordenadas son con respecto a la esquina superior izquierda del `PanelAutoContenido`.

```
1  // Fig. 14.4: PanelAutoContenido.java
2  // Una clase JPanel autocontenida que maneja sus propios eventos de ratón.
3  package com.deitel.cpej5.cap14;
4
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class PanelAutoContenido extends JPanel {
10    private int x1, y1, x2, y2;
11
12    // establecer manejo de eventos de ratón para PanelAutoContenido
13    public PanelAutoContenido()
14    {
15       // establecer componente de escucha de eventos de ratón
16       addMouseListener(
17
18          new MouseAdapter() {  // clase interna anónima
19
20             // manejar evento de oprimir botón del ratón
21             public void mousePressed( MouseEvent evento )
22             {
23                x1 = evento.getX();
24                y1 = evento.getY();
25             }
26
27             // manejar evento de soltar botón del ratón
28             public void mouseReleased( MouseEvent evento )
29             {
30                x2 = evento.getX();
31                y2 = evento.getY();
32                repaint();
33             }
34
35          } // fin de la clase interna anónima
36
37       ); // fin de la llamada a addMouseListener
38
39       // establecer componente de escucha de movimientos del ratón
40       addMouseMotionListener(
41
```

http://librosysolucionarios.net

Figura 14.4 Subclase de `JPanel` que procesa eventos de ratón. (Parte 1 de 2.)


```
42          new MouseMotionAdapter() {  // clase interna anónima
43
44             // manejar evento de arrastrar el ratón
45             public void mouseDragged( MouseEvent evento )
46             {
47                x2 = evento.getX();
48                y2 = evento.getY();
49                repaint();
50             }
51
52          } // fin de la clase interna anónima
53
54       ); // fin de la llamada a addMouseMotionListener
55
56    } // fin del constructor de PanelAutoContenido
57
58    // devolver anchura y altura preferidas de PanelAutoContenido
59    public Dimension getPreferredSize()
60    {
61       return new Dimension( 150, 100 );
62    }
63
64    // pintar un óvalo en las coordenadas especificadas
65    public void paintComponent( Graphics g )
66    {
67       super.paintComponent( g );
68
69       g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
70          Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
71    }
72
73 } // fin de la clase PanelAutoContenido
```

Figura 14.4 Subclase de `JPanel` que procesa eventos de ratón. (Parte 2 de 2.)

Observación de apariencia visual 14.5

La acción de dibujar en cualquier componente de la GUI se realiza con coordenadas que se miden desde la esquina superior izquierda (0, 0) de ese componente de la GUI.

El constructor de `PanelAutoContenido` (líneas 13 a 56) utiliza los métodos `addMouseListener` y `addMouseMotionListener` para registrar objetos de clase interna que manejen los eventos de ratón y los eventos de movimiento de ratón para el objeto `PanelAutoContenido`. Sólo se sobrescriben `mousePressed` (líneas 21 a 25), `mouseReleased` (líneas 28 a 33) y `mouseDragged` (líneas 45 a 50) para realizar las tareas. Las clases internas heredan los demás métodos manejadores de ratón de las clases adaptadoras `MouseAdapter` y `MouseMotionAdapter`.

Al extender la clase `JPanel`, estamos creando un nuevo componente de la GUI. Por lo tanto, debemos asegurarnos que nuestro nuevo componente funcione como los demás componentes, para fines de su distribución en pantalla. A menudo, los administradores de esquemas utilizan el método `getPreferredSize` de un componente de la GUI (heredado de la clase `java.awt.Component`) para determinar la anchura y altura preferidas del componente, cuando éste se distribuya en pantalla como parte de una GUI. Si un nuevo componente tiene una anchura y altura preferidas, debe sobrescribir el método `getPreferredSize` (líneas 59 a 62) para devolver esa anchura y altura como un objeto de la clase `Dimension` (paquete `java.awt`).

Observación de apariencia visual 14.6

El tamaño predeterminado de un objeto `JPanel` es 10 píxeles de ancho y 10 de alto.

Observación de apariencia visual 14.7

Al derivar subclases de `JPanel` (o de cualquier otra subclase de `JComponent`), se debe sobrescribir el método `getPreferredSize` si el nuevo componente debe tener una anchura y alturas preferidas específicas.

El método `paintComponent` (líneas 65 a 71) dibuja un óvalo, utilizando los valores actuales de las variables de instancia `x1`, `y1`, `x2` y `y2`. La esquina superior izquierda, anchura y altura se determinan oprimiendo y soltando el botón del ratón, arrastrando el ratón y soltándolo en el objeto `PanelAutoContenido`.

Las coordenadas iniciales `x1` y `y1` en el área de dibujo de `PanelAutoContenido` se capturan en el método `mousePressed` (líneas 21 a 25). A medida que el usuario arrastra el ratón, el programa genera una serie de llamadas a `mouseDragged` (líneas 45 a 50) mientras el usuario continúa sosteniendo el botón del ratón y moviéndolo. Cada llamada captura en las variables `x2` y `y2` la ubicación actual del ratón, con respecto a la esquina superior izquierda del `PanelAutoContenido` y llama a `repaint` para dibujar la versión actual del óvalo. La acción de dibujar se confina estrictamente al `PanelAutoContenido`, incluso si el usuario arrastra el ratón fuera del área de dibujo de `PanelAutoContenido`. Todo lo que se dibuje fuera del `PanelAutoContenido` se *corta*; no se muestran píxeles fuera de los límites del `PanelAutoContenido`.

Los cálculos que se proporcionan en el método `paintComponent` (líneas 69 a 70) determinan la esquina superior izquierda apropiada, utilizando el método `Math.min` dos veces para encontrar los valores más pequeños para las coordenadas *x* y *y*. La anchura y altura del óvalo deben ser valores positivos o de lo contrario el óvalo no se mostrará en pantalla. El método `Math.abs` obtiene el valor absoluto de los cálculos `x1 - x2` y `y1 - y2` que determinan la anchura y altura, respectivamente, del rectángulo delimitador del óvalo. Cuando se completan los cálculos, `paintComponent` dibuja el óvalo. La llamada a la versión de `paintComponent` de la superclase al principio del método asegura que el óvalo anterior que aparezca en el `PanelAutoContenido` se borre, antes de mostrar el nuevo.

Observación de apariencia visual 14.8

La mayoría de los componentes de la GUI de Swing pueden ser transparentes u opacos. Si un componente de la GUI de Swing es opaco, cuando se haga una llamada a su método `paintComponent` se borrará su fondo. En caso contrario, su fondo no se borrará. Sólo los componentes opacos pueden mostrar un color de fondo personalizado.

Observación de apariencia visual 14.9

Los objetos `JPanel` son opacos de manera predeterminada.

Cuando el usuario suelta el botón del ratón, el método `mouseReleased` (líneas 28 a 33) captura en las variables `x2` y `y2` la ubicación final del ratón, e invoca a `repaint` para dibujar la versión final del óvalo.

`PruebaPanelAutoContenido` muestra un objeto `PanelAutoContenido`, en el cual el usuario puede arrastrar el ratón para dibujar. El constructor (líneas 14 a 50 de la figura 14.5) crea un `PanelAutoContenido` (línea 17) y establece su color de fondo (línea 18) en amarillo, de manera que su área contraste con el fondo de la ventana de la aplicación.

Nos gustaría que este programa diferenciara entre los eventos de movimiento del ratón en el `PanelAutoContenido` y los eventos de movimiento de ratón en la ventana de la aplicación, por lo que en las líneas 25 a 45 se registra un manejador de eventos para los eventos de movimiento de ratón de la ventana de la aplicación. Los manejadores de eventos `mouseDragged` (líneas 30 a 34) y `mouseMoved` (líneas 37 a 41) utilizan el método `setTitle` (heredado indirectamente de la clase `java.awt.Frame`) para mostrar una cadena en la barra de título de la ventana, que indique la coordenada *x*-*y* en donde ocurrió el evento de movimiento del ratón.

```
1  // Fig. 14.5: PruebaPanelAutoContenido.java
2  // Creación de una subclase autocontenida de JPanel que procesa
3  // sus propios eventos de ratón.
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
```

Figura 14.5 Captura de los eventos de ratón con un objeto `JPanel`. (Parte 1 de 3.)


```
7
8  import com.deitel.cpej5.capi4.PanelAutoContenido;
9
10 public class PruebaPanelAutoContenido extends JFrame {
11    private PanelAutoContenido miPanel;
12
13    // configurar la GUI y manejadores de eventos de movimiento del ratón para la
14    ventana de la aplicación public PruebaPanelAutoContenido()
15    {
16       // establecer un PanelAutoContenido
17       miPanel = new PanelAutoContenido();
18       miPanel.setBackground( Color.YELLOW );
19
20       Container contenedor = getContentPane();
21       contenedor.setLayout( new FlowLayout() );
22       contenedor.add( miPanel );
23
24       // establecer manejo de eventos de movimiento del ratón
25       addMouseMotionListener(
26
27          new MouseMotionListener() {  // clase interna anónima
28
29             // manejar evento de arrastrar el ratón
30             public void mouseDragged( MouseEvent evento )
31             {
32                setTitle( "Arrastrando: x=" + evento.getX() +
33                   "; y=" + evento.getY() );
34             }
35
36             // manejar evento de mover el ratón
37             public void mouseMoved( MouseEvent evento )
38             {
39                setTitle( "Moviendo: x=" + evento.getX() +
40                   "; y=" + evento.getY() );
41             }
42
43          } // fin de la clase interna anónima
44
45       ); // fin de la llamada a addMouseMotionListener
46
47       setSize( 300, 200 );
48       setVisible( true );
49
50    } // fin del constructor de PruebaPanelAutoContenido
51
52    public static void main( String args[] )
53    {
54       PruebaPanelAutoContenido aplicacion = new PruebaPanelAutoContenido();
55       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
56    }
57
58 } // fin de la clase PruebaPanelAutoContenido
```

Figura 14.5 Captura de los eventos de ratón con un objeto `JPanel`. (Parte 2 de 3.)

http://librosysolucionarios.net

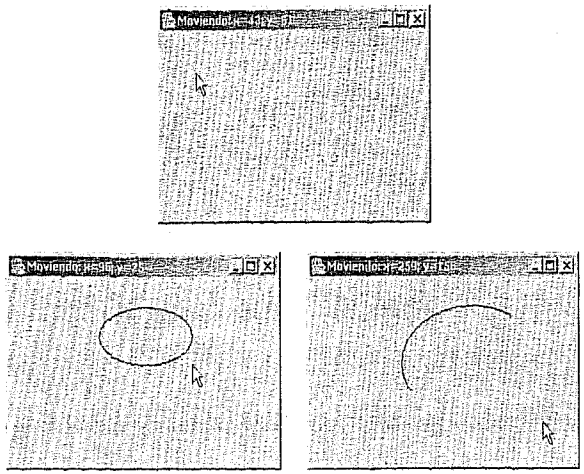


Figura 14.5 Captura de los eventos de ratón con un objeto `JPanel`. (Parte 3 de 3.)

Al ejecutar este programa, pruebe a arrastrar el ratón desde el fondo de la ventana de aplicación hacia el área del `PanelAutoContenido`, para ver que los eventos de arrastre se envíen a la ventana de aplicación, en vez de que se envíen al `PanelAutoContenido`. Luego, empiece una nueva operación de arrastre en el área del `PanelAutoContenido` y arrastre el ratón hacia el fondo de la ventana de aplicación, para ver que los eventos de arrastre se envíen al `PanelAutoContenido`, en vez de enviarse a la ventana de aplicación. Recuerde que en la sección 13.12 vimos que los eventos de arrastre del ratón se envían al componente en el que se haya iniciado la acción de arrastrar el ratón.

Observación de apariencia visual 14.10

Una operación de arrastre del ratón empieza con un evento de oprimir un botón del ratón. Todos los eventos de arrastre de ratón subsiguientes (hasta que el usuario suelte el botón del ratón) se envían al componente de la GUI que recibió el evento original de oprimir un botón del ratón.

14.5 `JSlider`

Los objetos ***`JSlider`*** permiten al usuario seleccionar de entre un rango de valores enteros. La clase ***`JSlider`*** hereda de `JComponent`. En la figura 14.6 se muestra un objeto `JSlider` horizontal con *marcas* y el *indicador* que permite al usuario seleccionar un valor. Los objetos `Jslider` pueden personalizarse para mostrar *marcas más distanciadas*, *marcas menos distanciadas* y etiquetas para las marcas. También soportan el *ajuste a la marca*, en el que al colocar el indicador entre dos marcas, éste se *ajusta* a la marca más cercana.

La mayoría de los componentes de la GUI de Swing soportan las interacciones del usuario mediante el ratón y el teclado. Por ejemplo, si un objeto `JSlider` tiene el enfoque (es decir, que sea el componente de la GUI seleccionado en ese momento, en la interfaz de usuario), la *tecla de flecha izquierda* y la *tecla de flecha derecha* hacen que el indicador del objeto `JSlider` se decremente o se incremente en 1, respectivamente. La



Figura 14.6 Componente `JSlider` con orientación horizontal.

http://librosysolucionarios.net

tecla de flecha hacia abajo y la *tecla de flecha hacia arriba* también hacen que el indicador del objeto `JSlider` se decremente o incremente en 1 marca, respectivamente. Las teclas *Av Pág* (avance de página) y *Re Pág* (retroceso de página) hacen que el indicador del objeto `JSlider` se decremente o incremente en *incrementos de bloque* de una décima parte del rango de valores, respectivamente. La *tecla Inicio* desplaza el indicador hacia el valor mínimo del objeto `JSlider`, y la *tecla Fin* desplaza el indicador hacia el valor máximo del objeto `JSlider`.

Los objetos `JSlider` tienen una *orientación horizontal* o una *orientación vertical*. Para un objeto `JSlider` horizontal, el valor mínimo se encuentra en el extremo izquierdo y el valor máximo se encuentra en el extremo derecho. Para un objeto `JSlider` vertical, el valor mínimo se encuentra en el extremo inferior y el valor máximo se encuentra en el extremo superior. La posición relativa del indicador muestra el valor actual del objeto `JSlider`.

Observación de apariencia visual 14.11

Las posiciones de los valores mínimo y máximo en un objeto `JSlider` pueden invertirse, invocando al método `setInverted` de `JSlider` con el argumento `boolean` `true`.

El programa de las figuras 14.7 y 14.8 permite al usuario ajustar el tamaño de un círculo dibujado en una subclase de `JPanel`, llamada `PanelOvalo` (figura 14.7). El usuario especifica el diámetro del círculo con un objeto `JSlider` horizontal. La clase de aplicación `DemoSlider` (figura 14.8) crea el objeto `JSlider` que controla el diámetro del círculo. La clase `PanelOvalo` es una subclase de `JPanel` que sabe cómo dibujar un círculo en sí misma, utilizando su propia variable de instancia `diametro` para determinar el diámetro del círculo; el `diametro` se utiliza como la anchura y altura del cuadro delimitador en el que se muestra el círculo. El valor del `diametro` se establece cuando el usuario interactúa con el objeto `JSlider`. El manejador de eventos llama al método `establecerDiametro` en la clase `PanelOvalo` para establecer el `diametro`, y llama a `repaint` para dibujar el nuevo círculo. La llamada a `repaint` resulta en una llamada al método `paintComponent` de `PanelOvalo`.

La clase `PanelOvalo` (figura 14.7) contiene un método `paintComponent` (líneas 10 a 15) que dibuja un óvalo relleno (un círculo, en este ejemplo), un método `establecerDiametro` (líneas 18 a 23) que modifica el diámetro del círculo y pinta nuevamente (mediante `repaint`) el `PanelOvalo`, un método `getPreferredSize` (líneas 26 a 29) que devuelve el valor predeterminado de la anchura y altura preferidas de un `PanelOvalo`, y un método `getMinimumSize` (líneas 32 a 35) que devuelve el valor predeterminado de la anchura y altura mínimas de un `PanelOvalo`.

Observación de apariencia visual 14.12

Si un nuevo componente de la GUI tiene una anchura y altura mínimas (es decir, que unas medidas menores hagan que el componente se muestre incorrectamente en la pantalla), debe sobrescribir el método `getMinimumSize` para devolver la anchura y altura mínimas como una instancia de la clase `Dimension`.

Observación de apariencia visual 14.13

Para muchos componentes de la GUI, el método `getMinimumSize` se implementa para devolver el resultado de una llamada al método `getPreferredSize` de ese componente.

El constructor de la clase `DemoSlider` (líneas 13 a 50 de la figura 14.8) crea el objeto `PanelOvalo` llamado `miPanel` (línea 18) y establece su color de fondo (línea 19). En las líneas 22 y 23 se crea el objeto `JSlider` llamado `sliderDiametro` para controlar el diámetro del círculo que se dibuja en el `PanelOvalo`. La orientación de `sliderDiametro` es `HORIZONTAL` (una constante en la interfaz `SwingConstants`). El segundo y tercer argumentos del constructor de `JSlider` indican los valores enteros máximo y mínimo en el rango de valores para este objeto `JSlider`. El último argumento del constructor indica que el valor inicial del objeto `JSlider` (es decir, en dónde se mostrará el indicador) debe ser `10`.

En las líneas 24 a 25 se personaliza la apariencia del objeto `JSlider`. El método ***`setMajorTickSpacing`*** indica que cada marca principal representa 10 valores, en el rango de valores soportado por el objeto `JSlider`. El método ***`setPaintTicks`*** con un argumento `true` indica que deben mostrarse las marcas (éstas no se muestran de manera predeterminada). Para ver otros métodos utilizados para personalizar la apariencia de un objeto `JSlider`, consulte la documentación en línea para el objeto `JSlider` (`java.sun.com/j2se/1.4.1/docs/api/javax/swing/JSlider.html`).

```
1  // Fig. 14.7: PanelOvalo.java
2  // Una clase JPanel personalizada.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class PanelOvalo extends JPanel {
7     private int diametro = 10;
8
9     // dibujar un óvalo del diámetro especificado
10    public void paintComponent( Graphics g )
11    {
12       super.paintComponent( g );
13
14       g.fillOval( 10, 10, diametro, diametro );
15    }
16
17    // validar y establecer el diámetro, después pintar nuevamente
18    public void establecerDiametro( int nuevoDiametro )
19    {
20       // si el diámetro es inválido, usar valor predeterminado de 10
21       diametro = ( nuevoDiametro >= 0 ? nuevoDiametro : 10 );
22       repaint();
23    }
24
25    //utilizado por el administrador de esquemas para determinar el tamaño preferido
26    public Dimension getPreferredSize()
27    {
28       return new Dimension( 200, 200 );
29    }
30
31    //utilizado por el administrador de esquemas para determinar el tamaño mínimo
32    public Dimension getMinimumSize()
33    {
34       return getPreferredSize();
35    }
36
37 } // fin de la clase PanelOvalo
```

Figura 14.7 Subclase de `JPanel` para dibujar círculos de un diámetro especificado.

```
1  // Fig. 14.8: DemoSlider.java
2  // Uso de objetos JSlider para ajustar el tamaño de un óvalo.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.event.*;
7
8  public class DemoSlider extends JFrame {
9     private JSlider sliderDiametro;
10    private PanelOvalo miPanel;
11
12    // configurar la GUI
13    public DemoSlider()
14    {
15       super( "Demostración de JSlider" );
```



```
16
17       // establecer PanelOvalo
18       miPanel = new PanelOvalo();
19       miPanel.setBackground( Color.YELLOW );
20
21       // establecer objeto JSlider para controlar el valor del diámetro
22       sliderDiametro =
23          new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
24       sliderDiametro.setMajorTickSpacing( 10 );
25       sliderDiametro.setPaintTicks( true );
26
27       // registrar componente de escucha de eventos de JSlider
28       sliderDiametro.addChangeListener(
29
30          new ChangeListener() { // clase interna anónima
31
32             // manejar cambio en el valor del control deslizable
33             public void stateChanged( ChangeEvent e )
34             {
35                miPanel.establecerDiametro( sliderDiametro.getValue() );
36             }
37
38          } // fin de la clase interna anónima
39
40       ); // fin de la llamada a addChangeListener
41
42       // adjuntar componentes al panel de contenido
43       Container contenedor = getContentPane();
44       contenedor.add( sliderDiametro, BorderLayout.SOUTH );
45       contenedor.add( miPanel, BorderLayout.CENTER );
46
47       setSize( 220, 270 );
48       setVisible( true );
49
50    } // fin del constructor de DemoSlider
51
52    public static void main( String args[] )
53    {
54       DemoSlider aplicacion = new DemoSlider();
55       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
56    }
57
58 } // fin de la clase DemoSlider
```

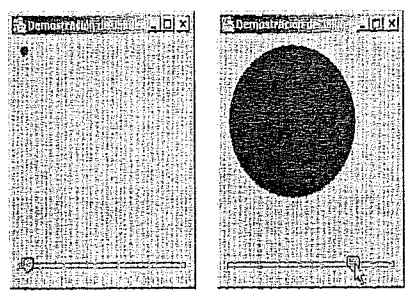


Figura 14.8 Valor de `JSlider` utilizado para determinar el diámetro de un círculo. (Parte 2 de 2.)

http://librosysolucionarios.net

Los objetos `JSlider` generan eventos `ChangeEvent` (paquete `javax.swing.event`) en respuesta a las interacciones de los usuarios. Un objeto de una clase que implementa a la interfaz `ChangeListener` (paquete `javax.swing.event`) y declara al método `stateChanged`, puede responder a eventos `ChangeEvent`. En las líneas 28 a 40 se registra un objeto `ChangeListener` para manejar los eventos de `sliderDiametro`. Cuando se llama al método `stateChanged` (líneas 33 a 36) en respuesta a la interacción de un usuario, en la línea 35 se hace una llamada al método `setDiameter` de `miPanel` y se le pasa el valor actual del objeto `JSlider` como un argumento. El método `getValue` de `JSlider` devuelve la posición actual del indicador.

14.6 Ventanas: Observaciones adicionales

Desde el capítulo 10 hasta este capítulo, la mayoría de las aplicaciones han utilizado subclases de `JFrame` como ventanas de aplicación. En esta sección, hablaremos sobre algunas cuestiones importantes de `JFrame`.

Un objeto `JFrame` es una *ventana* con una *barra de título* y un *borde*. La clase `JFrame` es una subclase de `java.awt.Frame` (que es una subclase de `java.awt.Window`). Como tal, `JFrame` es uno de los pocos componentes de la GUI de Swing que no es un componente de la GUI ligero. Cuando se muestra una ventana desde un programa de Java, ésta se proporciona mediante el kit de herramientas de ventana de la plataforma local; la ventana tendrá la misma apariencia que las otras ventanas que se muestren en esa plataforma. Cuando se ejecute un programa de Java en una Macintosh, en donde se muestre una ventana, la barra de título y los bordes de la ventana tendrán la misma apariencia que las ventanas de las demás aplicaciones Macintosh. Cuando se ejecute un programa de Java en Microsoft Windows, en donde se muestre una ventana, la barra de título y los bordes tendrán la misma apariencia que las ventanas de las demás aplicaciones Microsoft Windows. Y cuando se ejecute un programa de Java en una plataforma UNIX, en donde se muestre una ventana, la barra de título y los bordes de la ventana tendrán la misma apariencia que las ventanas de las demás aplicaciones UNIX en esa plataforma.

La clase `JFrame` soporta tres operaciones cuando el usuario cierra la ventana. De manera predeterminada, una ventana se oculta (es decir, se quita de la pantalla) cuando el usuario la cierra. Esto puede controlarse con el método `setDefaultCloseOperation` de `JFrame`. La interfaz `WindowConstants` (paquete `javax.swing`), que es implementada por la clase `JFrame`, declara tres constantes para ser utilizadas con este método: `DISPOSE_ON_CLOSE`, `DO_NOTHING_ON_CLOSE` y `HIDE_ON_CLOSE` (el valor predeterminado). Algunas plataformas sólo permiten que se muestre un número limitado de ventanas en la pantalla. Por lo tanto, una ventana es un valioso recurso que debe regresarse al sistema cuando ya no se necesite. La clase `Window` (una superclase indirecta de `JFrame`) declara el método `dispose` para este fin. Cuando ya no se necesita un objeto `Window` en una aplicación, se debe desechar explícitamente este objeto `Window`. Esto puede hacerse llamando al método `dispose` de `Window`, o llamando al método `setDefaultCloseOperation` con el argumento `WindowConstants.DISPOSE_ON_CLOSE`. Aparte, si se termina una aplicación se devuelven todos los recursos de ventana al sistema. Al establecer la operación de cierre predeterminada en `DO_NOTHING_ON_CLOSE`, el programa determinará lo que debe hacer cuando el usuario le indique que debe cerrar la ventana. Recuerde que en la sección 10.9 vimos que los objetos `JFrame` también soportan el uso de la constante `EXIT_ON_CLOSE` como argumento para el método `setDefaultCloseOperation`, para especificar que la aplicación debe terminar cuando el usuario cierre la ventana.



Buena práctica de programación 14.1

Las ventanas son un recurso imprescindible del sistema. Devuélvalas al sistema cuando ya no sean necesarias.

De manera predeterminada, una ventana no se muestra en la pantalla sino hasta que el programa invoque al método `setVisible` de esta ventana (heredado de la clase `java.awt.Component`) con un argumento `true`. Además, debe establecerse el tamaño de una ventana mediante una llamada al método `setSize` (heredado de la clase `java.awt.Component`). La posición que tendrá la ventana cuando aparezca en la pantalla se especifica mediante el método `setLocation` (heredado de la clase `java.awt.Component`).



Error común de programación 14.3

Olvidar llamar al método `setVisible` en una ventana es un error lógico en tiempo de ejecución; la ventana no se mostrará en pantalla.



Error común de programación 14.4

Olvidar llamar al método `setSize` en una ventana es un error lógico en tiempo de ejecución; sólo aparecerá la barra de título.

Las ventanas generan *eventos de ventana* cuando el usuario las manipula. Los componentes de escucha de eventos se registran para eventos de ventana mediante el método `addWindowListener` de la clase `Window`. La interfaz `WindowListener` proporciona siete métodos manejadores de eventos de ventana: `windowActivated` (se invoca cuando el usuario hace que una ventana sea la ventana activa), `windowClosed` (se invoca cuando se cierra la ventana), `windowClosing` (se invoca cuando el usuario inicia el cierre de la ventana), `windowDeactivated` (se invoca cuando el usuario hace que otra ventana sea la ventana activa), `windowIconified` (se invoca cuando el usuario minimiza una ventana), `windowDeiconified` (se invoca cuando el usuario restaura una ventana, después de que ha sido minimizada) y `windowOpened` (se invoca cuando un programa muestra por primera vez una ventana en la pantalla).

14.7 Uso de menús con marcos

Los menús son una parte integral de las GUIs. Permiten al usuario realizar acciones sin "atestar" innecesariamente una GUI con componentes adicionales. En las GUIs de Swing, los menús pueden adjuntarse solamente a los objetos de clases que proporcionen el método `setJMenuBar`. Dos de esas clases son `JFrame` y `JApplet`. Las clases utilizadas para declarar menús son `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` y la clase `JRadioButtonMenuItem`.



Observación de apariencia visual 14.14

Los menús simplifican las GUIs al reducir el número de componentes que el usuario puede ver a la vez.

La clase `JMenuBar` (una subclase de `JComponent`) contiene los métodos necesarios para administrar una *barra de menús*, la cual es un contenedor para los menús. La clase `JMenu` (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar *menús*. Los menús contienen elementos de menú, y pueden agregarse a barras de menús o a otros menús, como submenús. Cuando se hace clic en un menú, éste se expande para mostrar su lista de elementos de menú.

La clase `JMenuItem` (una subclase de `javax.swing.AbstractButton`) contiene los métodos necesarios para administrar *elementos de menú*. Un elemento de menú es un componente de la GUI que está dentro de un menú y que, cuando se selecciona, produce un evento de acción. Un elemento de menú puede usarse para iniciar una acción, o puede ser un *submenú* que proporcione más elementos de menú que el usuario pueda seleccionar. Los submenús son útiles para agrupar en un menú los elementos de menú que estén relacionados.

La clase `JCheckBoxMenuItem` (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar los elementos de menú que pueden activarse o desactivarse. Cuando se selecciona un objeto `JCheckBoxMenuItem`, aparece una marca de verificación a la izquierda del elemento de menú. Cuando se selecciona de nuevo el mismo objeto `JCheckBoxMenuItem`, se quita la marca de verificación a la izquierda del elemento de menú.

La clase `JRadioButtonMenuItem` (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar elementos de menú que pueden activarse o desactivarse, de manera parecida a los objetos `JCheckBoxMenuItem`. Cuando se mantienen varios objetos `JRadioButtonMenuItem` como parte de un objeto `ButtonGroup`, sólo puede seleccionarse un elemento en el grupo a la vez. Cuando se selecciona un objeto `JRadioButtonMenuItem`, aparece un círculo relleno a la izquierda del elemento de menú. Cuando se selecciona otro objeto `JRadioButtonMenuItem`, se quita el círculo relleno a la izquierda del elemento de menú previamente seleccionado.

La aplicación de la figura 14.9 demuestra el uso de varios elementos de menú. Este programa también demuestra cómo especificar caracteres especiales (llamados *nemónicos*) que pueden proporcionar un acceso rápido a un menú o elemento de menú, desde el teclado. Los nemónicos pueden usarse con todas las subclases de `javax.swing.AbstractButton`.

La clase `PruebaMenu` (línea 7) declara a los componentes de la GUI y el manejo de eventos para los elementos de menú. La mayor parte del código en esta aplicación aparece en el constructor de la clase (líneas 17 a 150).

http://librosysolucionarios.net

```
1  // Fig. 14.9: PruebaMenu.java
2  // Demostración de los menús
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaMenu extends JFrame {
8     private final Color valoresColor[] =
9        { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
10    private JRadioButtonMenuItem elementosColor[], tiposLetra[];
11    private JCheckBoxMenuItem elementosEstilo[];
12    private JLabel pantallaEtiqueta;
13    private ButtonGroup grupoTiposLetra, grupoColores;
14    private int estilo;
15
16    // configurar la GUI
17    public PruebaMenu()
18    {
19       super( "Uso de objetos JMenu" );
20
21       // establecer menú Archivo y sus elementos de menú
22       JMenu menuArchivo = new JMenu( "Archivo" );
23       menuArchivo.setMnemonic( 'A' );
24
25       // establecer elemento de menú Acerca de...
26       JMenuItem elementoAcerca = new JMenuItem( "Acerca de..." );
27       elementoAcerca.setMnemonic( 'c' );
28       menuArchivo.add( elementoAcerca );
29       elementoAcerca.addActionListener(
30
31          new ActionListener() {  // clase interna anónima
32
33             // mostrar cuadro de diálogo de mensaje cuando el usuario seleccione
                  Acerca de...
34             public void actionPerformed( ActionEvent evento )
35             {
36                JOptionPane.showMessageDialog( PruebaMenu.this,
37                   "Éste es un ejemplo\ndel uso de menús",
38                   "Acerca de", JOptionPane.PLAIN_MESSAGE );
39             }
40
41          }  // fin de la clase interna anónima
42
43       ); // fin de la llamada a addActionListener
44
45       // establecer elemento de menú Salir
46       JMenuItem elementoSalir = new JMenuItem( "Salir" );
47       elementoSalir.setMnemonic( 'S' );
48       menuArchivo.add( elementoSalir );
49       elementoSalir.addActionListener(
50
51          new ActionListener() {  // clase interna anónima
52
53             // terminar la aplicación cuando el usuario haga clic en elementoSalir
54             public void actionPerformed( ActionEvent evento )
55             {
```

Figura 14.9 Los objetos **JMenu** y los nemónicos. (Parte 1 de 5.)


```
56                System.exit( 0 );
57             }
58
59          }  // fin de la clase interna anónima
60
61       ); // fin de la llamada a addActionListener
62
63       // crear barra de menús y adjuntarla a la ventana PruebaMenu
64       JMenuBar barra = new JMenuBar();
65       setJMenuBar( barra );
66       barra.add( menuArchivo );
67
68       // crear menú Formato, con sus submenús y elementos de menú
69       JMenu menuFormato = new JMenu( "Formato" );
70       menuFormato.setMnemonic( 'F' );
71
72       // crear submenú Color
73       String colores[] = { "Negro", "Azul", "Rojo", "Verde" };
74
75       JMenu menuColor = new JMenu( "Color" );
76       menuColor.setMnemonic( 'C' );
77
78       elementosColor = new JRadioButtonMenuItem[ colores.length ];
79       grupoColores = new ButtonGroup();
80       ManejadorEventos manejadorEventos = new ManejadorEventos();
81
82       // crear elementos de menú tipo botones de opción para el menú Color
83       for ( int cuenta = 0; cuenta < colores.length; cuenta++ ) {
84          elementosColor[ cuenta ] =
85             new JRadioButtonMenuItem( colores[ cuenta ] );
86          menuColor.add( elementosColor[ cuenta ] );
87          grupoColores.add( elementosColor[ cuenta ] );
88          elementosColor[ cuenta ].addActionListener( manejadorEventos );
89       }
90
91       // seleccionar primer elemento del menú Color
92       elementosColor[ 0 ].setSelected( true );
93
94       // agregar el menú Formato a la barra de menús
95       menuFormato.add( menuColor );
96       menuFormato.addSeparator();
97
98       // crear submenú Tipo de letra
99       String nombresTiposLetra[] = { "Serif", "Monospaced", "SansSerif" };
100
101      JMenu menuTiposLetra = new JMenu( "Tipo de letra" );
102      menuTiposLetra.setMnemonic( 'T' );
103
104      tiposLetra = new JRadioButtonMenuItem[ nombresTiposLetra.length ];
105      grupoTiposLetra = new ButtonGroup();
106
107      // crear elementos de menú tipo botones de opción para el menú Tipos de letra
108      for ( int cuenta = 0; cuenta < tiposLetra.length; cuenta++ ) {
109         tiposLetra[ cuenta ] = new JRadioButtonMenuItem( nombresTiposLetra[ cuenta ] );
110         menuTiposLetra.add( tiposLetra[ cuenta ] );
```

Figura 14.9 Los objetos **JMenu** y los nemónicos. (Parte 2 de 5.)

http://librosysolucionarios.net

```
111            grupoTiposLetra.add( tiposLetra[ cuenta ] );
112            tiposLetra[ cuenta ].addActionListener( manejadorEventos );
113         }
114
115         // seleccionar el primer elemento del menú Tipo de letra
116         tiposLetra[ 0 ].setSelected( true );
117
118         menuTiposLetra.addSeparator();
119
120         // establecer elementos del menú Estilo
121         String nombresEstilo[] = { "Negrita", "Cursiva" };
122
123         elementosEstilo = new JCheckBoxMenuItem[ nombresEstilo.length ];
124         ManejadorEstilo manejadorEstilo = new ManejadorEstilo();
125
126         // crear elementos de menú tipo casilla de verificación para el menú Estilo
127         for ( int cuenta = 0; cuenta < nombresEstilo.length; cuenta++ ) {
128            elementosEstilo[ cuenta ] =
129               new JCheckBoxMenuItem( nombresEstilo[ cuenta ] );
130            menuTiposLetra.add( elementosEstilo[ cuenta ] );
131            elementosEstilo[ cuenta ].addItemListener( manejadorEstilo );
132         }
133
134         // colocar menú Tipo de letra en el menú Formato
135         menuFormato.add( menuTiposLetra );
136
137         // agregar menú Formato a la barra de menús
138         barra.add( menuFormato );
139
140         // establecer etiqueta para mostrar texto
141         pantallaEtiqueta = new JLabel( "Texto de ejemplo", SwingConstants.CENTER );
142         pantallaEtiqueta.setForeground( valoresColor[ 0 ] );
143         pantallaEtiqueta.setFont( new Font( "Serif", Font.PLAIN, 72 ) );
144
145         getContentPane().setBackground( Color.CYAN );
146         getContentPane().add( pantallaEtiqueta, BorderLayout.CENTER );
147
148         setSize( 550, 200 );
149         setVisible( true );
150
151      } // fin del constructor
152
153      public static void main( String args[] )
154      {
155         PruebaMenu aplicacion = new PruebaMenu();
156         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
157      }
158
159      // clase interna para manejar eventos de acción de los elementos de menú
160      private class ManejadorEventos implements ActionListener {
161
162         // procesar selecciones de color y tipo de letra
163         public void actionPerformed( ActionEvent evento )
164         {
165            // procesar selección de color
```

http://librosysolucionarios.net

Figura 14.9 Los objetos **JMenu** y los nemónicos. (Parte 3 de 5.)


```
166            for ( int cuenta = 0; cuenta < elementosColor.length; cuenta++ )
167
168               if ( elementosColor[ cuenta ].isSelected() ) {
169                  pantallaEtiqueta.setForeground( valoresColor[ cuenta ] );
170                  break;
171               }
172
173            // procesar selección de tipo de letra
174            for ( int cuenta = 0; cuenta < tiposLetra.length; cuenta++ )
175
176               if ( evento.getSource() == tiposLetra[ cuenta ] ) {
177                  pantallaEtiqueta.setFont(
178                     new Font( tiposLetra[ cuenta ].getText(), estilo, 72 ) );
179                  break;
180               }
181
182            repaint();
183
184         } // fin del método actionPerformed
185
186      } // fin de la clase ManejadorEventos
187
188      // clase interna para manejar eventos de los elementos de menú tipo casilla de
         verificación
189      private class ManejadorEstilo implements ItemListener {
190
191         // procesar selecciones de estilo de tipo de letra
192         public void itemStateChanged( ItemEvent e )
193         {
194            estilo = 0;
195
196            // checar selección de negrita
197            if ( elementosEstilo[ 0 ].isSelected() )
198               estilo += Font.BOLD;
199
200            // checar selección de cursiva
201            if ( elementosEstilo[ 1 ].isSelected() )
202               estilo += Font.ITALIC;
203
204            pantallaEtiqueta.setFont(
205               new Font( pantallaEtiqueta.getFont().getName(), estilo, 72 ) );
206
207            repaint();
208         }
209
210      } // fin de la clase ManejadorEstilo
211
212   } // fin de la clase PruebaMenu
```



Figura 14.9 Los objetos **JMenu** y los nemónicos. (Parte 4 de 5.)



Figura 14.9 Los objetos `JMenu` y los nemónicos. (Parte 5 de 5.)

En las líneas 22 a 61 se establece el menú **Archivo** y se adjunta a la barra de menús. El menú **Archivo** contiene un elemento de menú **Acerca de...**, el cual muestra un cuadro de diálogo de mensaje al ser seleccionado, y un elemento de menú **Salir** que puede seleccionarse para terminar la aplicación.

En la línea 22 se crea el objeto `JMenu` y se pasa a su constructor la cadena `"Archivo"` como el nombre del menú. En la línea 23 se utiliza el método `setMnemonic` de `AbstractButton` (el cual se hereda en la clase `JMenu`) para indicar que `A` es el *nemónico* para este menú. Al oprimir las teclas *Alt* y *A* se abre el menú, así como cuando se hace clic en el menú con el ratón. En la GUI, el carácter nemónico en el nombre del menú se muestra subrayado. (Vea las capturas de pantalla de la figura 14.9.)

Observación de apariencia visual 14.15

Los nemónicos proporcionan un acceso rápido a los comandos de menú y de botón, por medio del teclado.

Observación de apariencia visual 14.16

Deben utilizarse distintos nemónicos para cada uno de los botones o elementos de menú. Generalmente, la primera letra en la etiqueta del elemento de menú o del botón es la que se utiliza como nemónico. Si varios botones o elementos de menú inician con la misma letra, seleccione la siguiente letra más prominente en el nombre (por ejemplo **u** *se selecciona comúnmente para el elemento de menú Guardar como, del menú* **Archivo**).

En las líneas 26 a 27 se crea el objeto `JMenuItem` `elementoAcerca` con el texto `"Acerca de..."` y se establece su nemónico en la letra `c`. (No se utiliza `A` porque esa letra es el nemónico del menú **Archivo**.) Este elemento de menú se agrega al objeto `menuArchivo` en la línea 28 mediante el método `add` de `JMenu`. Para tener acceso al elemento **Acerca de...** a través del teclado, oprima la tecla *Alt* y la letra `A` para abrir el menú **Archivo**, después oprima *c* para seleccionar el elemento de menú **Acerca de...**. En las líneas 29 a 43 se crea un objeto `ActionListener` para procesar el evento de acción del objeto `elementoAcerca`. En las líneas 36 a 38 se muestra un cuadro de diálogo de mensaje. En la mayoría de los usos anteriores de `showMessageDialog`, el primer argumento ha sido `null`. El propósito del primer argumento es especificar la *ventana padre* para el cuadro de diálogo. La ventana padre ayuda a determinar en dónde va a mostrarse el cuadro de diálogo. Si la ventana padre se especifica como `null`, el cuadro de diálogo aparece en el centro de la pantalla. Si la ventana padre no es `null`, el cuadro de diálogo aparece centrado sobre la ventana padre especificada. En este ejemplo, el programa especifica la ventana padre con `PruebaMenu.this`; la referencia `this` del objeto `PruebaMenu`. Al utilizar la referencia `this` en una clase interna, si se especifica la palabra `this` por sí sola, se hace referencia al objeto de la clase interna. Para hacer referencia a la referencia `this` del objeto de la clase externa, debe calificar a `this` con el nombre de la clase externa y un punto (`.`).

Los cuadros de diálogo generalmente son *modales*. Un *cuadro de diálogo modal* no permite el acceso a ninguna de las otras ventanas en la aplicación, sino hasta que se cierre. Los cuadros de diálogo que se muestran con la clase `JOptionPane` son cuadros de diálogo modales. La clase `JDialog` puede usarse para crear sus propios cuadros de diálogo modales o no modales.

En las líneas 46 a 61 se crea el elemento de menú `elementoSalir`, se establece su nemónico en `S`, se agrega a `menuArchivo` y se registra un objeto `ActionListener` que termina la aplicación cuando el usuario selecciona `elementoSalir`.

En las líneas 64 a 66 se crea el objeto `JMenuBar`, se adjunta a la ventana de aplicación mediante el método `setJMenuBar` de `JFrame` y se utiliza el método `add` de `JMenuBar` para adjuntar `menuArchivo` a la barra de menús.

http://librosysolucionarios.net

Error común de programación 14.5

Si olvida establecer la barra de menús mediante el método `setJMenuBar` de `JFrame`, la barra de menús no se mostrará en el objeto `JFrame`.

Observación de apariencia visual 14.17

Los menús generalmente aparecen de izquierda a derecha, en el orden en el que se agregan a un objeto `JMenuBar`.

En las líneas 69 a 70 se crea el menú llamado `menuFormato` y se establece su nemónico en `F`.

En las líneas 75 y 76 se crea el menú llamado `menuColor` (éste será un submenú en el menú **Formato**) y se establece su nemónico en `C`. En la línea 78 se crea el arreglo `JRadioButtonMenuItem` llamado `elementosColor`, el cual hace referencia a los elementos de menú de `menuColor`. En la línea 79 se crea el objeto `ButtonGroup` llamado `grupoColores`, el cual asegura que sólo se seleccione uno de los elementos de menú del submenú **Colores** en un momento dado. En la línea 80 se crea una instancia de la clase interna `ManejadorEventos` (declarada en las líneas 160 a 186) para responder a las selecciones del submenú **Color** y del submenú **Tipo de letra** (que se describiremos en breve). En la instrucción `for` de las líneas 83 a 89 se crea a cada uno de los objetos `JRadioButtonMenuItem` en el arreglo `elementosColor`, se agrega cada uno de los elementos de menú a `menuColor`, se agrega cada uno de los elementos de menú a `grupoColores` y se registra el objeto `ActionListener` para cada elemento de menú.

En la línea 92 se utiliza el método `setSelected` de `AbstractButton` para seleccionar el primer elemento en el arreglo `elementosColor`. En la línea 95 se agrega `menuColor` como un submenú de `menuFormato`. En la línea 96 se agrega una línea *separadora* (o separador) al menú. El separador aparece como una línea horizontal en el menú.

Observación de apariencia visual 14.18

Un submenú se crea agregando a un menú como elemento de otro menú. Cuando el ratón se coloca sobre un submenú (o cuando se oprime el nemónico de ese submenú), éste se expande para mostrar sus elementos de menú.

Observación de apariencia visual 14.19

Pueden agregarse separadores a un menú para agrupar los elementos de menú en forma lógica.

Observación de apariencia visual 14.20

Puede agregarse cualquier componente de la GUI ligero (es decir, un componente de cualquier subclase de la clase `JComponent`) a un objeto `JMenu` o `JMenuBar`.

En las líneas 101 a 118 se crean el submenú **Tipo de letra** y varios objetos `JRadioButtonMenuItem`, y se selecciona el primer elemento del arreglo `JRadioButtonMenuItem` llamado `tiposLetra`. En la línea 123 se crea un arreglo `JCheckBoxMenuItem` para representar a los elementos de menú que especifican los estilos negrita y cursiva para los tipos de letra. En la línea 124 se crea una instancia de la clase interna `ManejadorEstilos` (declarada en las líneas 189 a 210) para responder a los eventos de `JCheckBoxMenuItem`. La instrucción `for` de las líneas 127 a 132 crea a cada objeto `JCheckBoxMenuItem`, agrega cada uno de los elementos de menú a `menuTiposLetra` y registra el objeto `ItemListener` para cada elemento de menú. En la línea 135 se agrega `menuTiposLetra` como un submenú de `menuFormato`. En la línea 138 se agrega el `menuFormato` a la `barra` (de menús).

En las líneas 141 a 143 se crea un objeto `JLabel` para el cual los elementos del menú **Formato** controlan el tipo de letra, su color y estilo. El color inicial de primer plano se establece con el primer elemento del arreglo `valoresColor` (`Color.BLACK`) y el tipo de letra inicial se establece en `Serif` con estilo `PLAIN` y tamaño de `72` puntos. En la línea 145 se establece el color de fondo del panel de contenido de la ventana en cyan, y en la línea 146 se adjunta el objeto `JLabel` a la región `CENTER` del esquema `BorderLayout` del panel de contenido.

El método `actionPerformed` de `ItemHandler` (líneas 163 a 184) utiliza dos instrucciones `for` para determinar cuál fue el elemento de menú de tipo de letra o de color que generó el evento, y establece el tipo de letra o el color del objeto `JLabel` llamado `pantallaEtiqueta`, respectivamente. La condición `if` de la línea 168 utiliza el método `isSelected` de `AbstractButton` para determinar cuál fue el objeto `JRadioButtonMenuItem` seleccionado. La condición `if` de la línea 176 utiliza el método `getSource` del ob-

jeto evento para obtener una referencia al objeto `JRadioButtonMenuItem` que generó el evento. En la línea 178 se utiliza el método `getText` de `AbstractButton` para obtener el nombre del tipo de letra del elemento de menú.

El programa llama al método `itemStateChanged` de `ManejadorEstilo` (líneas 192 a 208) si el usuario selecciona un objeto `JCheckBoxMenuItem` en el `menuTiposLetra`. En las líneas 197 y 201 se determina si uno o ambos objetos `JCheckBoxMenuItems` se seleccionan, y se utiliza su estado combinado para determinar el nuevo estilo del tipo de letra.

14.8 JPopupMenu

Muchas de las aplicaciones de computadora de la actualidad proporcionan lo que se conoce como *menús contextuales sensibles al contexto*. En Swing, tales menús se crean con la clase `JPopupMenu` (una subclase de `JComponent`). Estos menús proporcionan opciones específicas al componente para el cual se generó el *evento de desencadenamiento del menú contextual*. En la mayoría de los sistemas, este evento de desencadenamiento ocurre cuando el usuario oprime y suelta el botón derecho del ratón.

Observación de apariencia visual 14.21



El evento de desencadenamiento del menú contextual es específico para cada plataforma. En la mayoría de las plataformas que utilizan un ratón con varios botones, el evento de desencadenamiento del menú contextual ocurre cuando el usuario hace clic con el botón derecho del ratón en un componente que tiene soporte para un menú contextual.

En la figura 14.10 se crea un objeto `JPopupMenu`, el cual permite al usuario seleccionar uno de tres colores y cambiar el color de fondo de la ventana. Cuando el usuario hace clic con el botón derecho del ratón en el fondo de la ventana `PruebaContextual`, aparece un objeto `JPopupMenu`, el cual contiene unos colores. Si el usuario hace clic en un objeto `JRadioButtonMenuItem` para seleccionar un color, el método `actionPerformed` de `ItemHandler` cambia el color de fondo del panel de contenido de la ventana.

```
1  // Fig. 14.10: PruebaContextual.java
2  // Demostración de los objetos JPopupMenu
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaContextual extends JFrame {
8     private JRadioButtonMenuItem elementos[];
9     private final Color valoresColor[] =
10       { Color.BLUE, Color.YELLOW, Color.RED };
11    private JPopupMenu menuContextual;
12
13    // configurar la GUI
14    public PruebaContextual()
15    {
16       super( "Uso de objetos JPopupMenu" );
17
18       ManejadorEventos manejador = new ManejadorEventos();
19       String colores[] = { "Azul", "Amarillo", "Rojo" };
20
21       // establecer menú contextual y sus elementos
22       ButtonGroup grupoColores = new ButtonGroup();
23       menuContextual = new JPopupMenu();
24       elementos = new JRadioButtonMenuItem[ 3 ];
25
26       // construir cada elemento de menú y agregarlo al menú contextual; además
```

Figura 14.10 Objeto `JPopupMenu` para seleccionar colores. (Parte 1 de 3.)


```
27       // permitir el manejo de eventos para cada elemento de menú
28       for ( int cuenta = 0; cuenta < elementos.length; cuenta++ ) {
29          elementos[ cuenta ] = new JRadioButtonMenuItem( colores[ cuenta ] );
30          menuContextual.add( elementos[ cuenta ] );
31          grupoColores.add( elementos[ cuenta ] );
32          elementos[ cuenta ].addActionListener( manejador );
33       }
34
35       getContentPane().setBackground( Color.WHITE );
36
37       // declarar un objeto MouseListener para la ventana que muestra un objeto
38       // JPopupMenu cuando ocurre el evento de desencadenamiento del menú contextual
39       addMouseListener(
40
41          new MouseAdapter() {  // clase interna anónima
42
43             // manejar evento de oprimir botón del ratón
44             public void mousePressed( MouseEvent evento )
45             {
46                checkForTriggerEvent( evento );
47             }
48
49             // manejar evento de soltar el botón del ratón
50             public void mouseReleased( MouseEvent evento )
51             {
52                checkForTriggerEvent( evento );
53             }
54
55             // determinar si evento debe desencadenar el menú contextual
56             private void checkForTriggerEvent( MouseEvent evento )
57             {
58                if ( evento.isPopupTrigger() )
59                   menuContextual.show(
60                      evento.getComponent(), evento.getX(), evento.getY() );
61             }
62
63          } // fin de la clase interna anónima
64
65       ); // fin de la llamada a addMouseListener
66
67       setSize( 300, 200 );
68       setVisible( true );
69
70    } // fin del constructor de PruebaContextual
71
72    public static void main( String args[] )
73    {
74       PruebaContextual aplicacion = new PruebaContextual();
75       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
76    }
77
78    // clase interna privada para manejar eventos de elemento de menú
79    private class ManejadorEventos implements ActionListener {
80
81       // procesar selecciones de elementos de menú
```

Figura 14.10 Objeto `JPopupMenu` para seleccionar colores. (Parte 2 de 3.)

http://librosysolucionarios.net

```
82        public void actionPerformed( ActionEvent evento )
83        {
84           // determinar cuál elemento de menú fue seleccionado
85           for ( int i = 0; i < elementos.length; i++ )
86              if ( evento.getSource() == elementos[ i ] ) {
87                 getContentPane().setBackground( valoresColor[ i ] );
88                 return;
89              }
90        }
91
92     } // fin de la clase interna privada ManejadorEventos
93
94  } // fin de la clase PruebaContextual
```

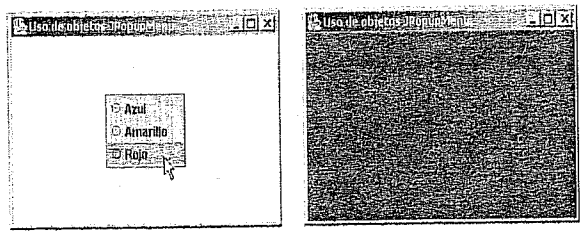


Figura 14.10 Objeto `JPopupMenu` para seleccionar colores. (Parte 3 de 3.)

En la línea 18 del constructor de `PruebaContextual` (líneas 14 a 70) se crea una instancia de la clase `ManejadorEventos` (declarada en las líneas 79 a 92), la cual procesará los eventos de los elementos de menú en el menú contextual. En la línea 23 se crea el objeto `JPopupMenu`. La instrucción `for` (líneas 28 a 33) crea un objeto `JRadioButtonMenuItem` (línea 29), lo agrega al objeto `menuContextual` (línea 30), agrega este objeto al objeto `ButtonGroup` llamado `grupoColores` (línea 31) para mantener sólo un objeto `JRadioButtonMenuItem` seleccionado a la vez, y registra su objeto `ActionListener` (línea 32). En la línea 35 se establece el fondo inicial en blanco, invocando al método `setBackground` del panel de contenido.

En las líneas 39 a 65 se registra un objeto `MouseListener` para manejar los eventos de ratón de la ventana de aplicación. Los métodos `mousePressed` (líneas 44 a 47) y `mouseReleased` (líneas 50 a 53) comprueban el evento de desencadenamiento del menú contextual. Cada método llama al método utilitario privado `checkForTriggerEvent` (líneas 56 a 61) para determinar si ocurrió el evento de desencadenamiento del menú contextual. El método `isPopupTrigger` de `MouseEvent` devuelve `true` si ocurrió el evento de desencadenamiento del menú contextual. De ser así, el método `show` de `JPopupMenu` muestra el objeto `JPopupMenu`. El primer argumento del método `show` especifica el *componente de origen*, cuya posición ayuda a determinar en dónde aparecerá objeto `JPopupMenu` en la pantalla. Los últimos dos argumentos son la coordenada *x-y* (medida desde la esquina superior izquierda del componente de origen) en donde debe aparecer el objeto `JPopupMenu`.

Observación de apariencia visual 14.22

Para mostrar un objeto `JPopupMenu` para el evento de desencadenamiento de menú contextual de varios componentes de la GUI, se deben registrar manejadores de eventos de ratón para cada uno de esos componentes de la GUI.

Cuando el usuario selecciona un elemento del menú contextual, el método `actionPerformed` de la clase `ManejadorEventos` (líneas 82 a 90) determina cuál objeto `JRadioButtonMenuItem` fue seleccionado por el usuario y establece el color de fondo del panel de contenido de la ventana.

14.9 Apariencia visual adaptable

Un programa que utiliza componentes de la GUI del Abstract Window Toolkit de Java (paquete `java.awt`) asume la apariencia visual de la plataforma en la que se ejecute. Un programa de Java ejecutándose en una Macintosh tiene la misma apariencia que los demás programas que se ejecutan en la Macintosh. Un programa ejecutándose en Microsoft Windows tiene la misma apariencia que los demás programas que se ejecutan en Microsoft Windows. Un programa de Java ejecutándose en una plataforma UNIX tiene la misma apariencia que los demás programas que se ejecutan en esa plataforma UNIX. Esto puede ser conveniente, ya que permite a los usuarios del programa utilizar, en cada plataforma, los componentes de la GUI con los que ya están familiarizados. Sin embargo, esto también introduce algunas cuestiones interesantes, relacionadas con la portabilidad.

Tip de portabilidad 14.1

Los componentes de la GUI en cada plataforma tienen distinta apariencia, lo cual puede requerir de distintas cantidades de espacio para mostrarse en pantalla. Esto podría cambiar la distribución y alineación de los componentes de la GUI.

Tip de portabilidad 14.2

Los componentes de la GUI en cada plataforma tienen distinta funcionalidad predeterminada (por ejemplo, algunas plataformas permiten que un botón con el enfoque se "oprima" mediante la barra de espaciamiento, mientras que otras no).

Los componentes de la GUI ligeros de Swing eliminan muchas de estas cuestiones, al proporcionar una funcionalidad uniforme entre plataformas, y al definir una apariencia visual uniforme entre plataformas (a la que se le conoce como la apariencia visual *metálica*). Swing también proporciona la flexibilidad de personalizar la apariencia visual, para que un programa tenga la apariencia visual al estilo Microsoft Windows (en sistemas Windows), al estilo Motif (UNIX) (en todas las plataformas) o al estilo Mac (sistemas Mac).

El programa de la figura 14.11 demuestra cómo cambiar la apariencia visual de una GUI de Swing. El programa crea varios componentes de la GUI, por lo que usted podrá ver el cambio en la apariencia visual de varios componentes de la GUI al mismo tiempo. La primera ventana de salida muestra la apariencia visual metálica estándar, la segunda ventana de salida muestra la apariencia visual Motif y la tercera ventana muestra la apariencia visual Windows.

```
1  // Fig. 14.11: DemoAparienciaVisual.java
2  // Cambio de la apariencia visual.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoAparienciaVisual extends JFrame {
8     private final String cadenas[] = { "Metal", "Motif", "Windows" };
9     private UIManager.LookAndFeelInfo apariencias[];
10    private JRadioButton opcion[];
11    private ButtonGroup grupo;
12    private JButton boton;
13    private JLabel etiqueta;
14    private JComboBox cuadroCombinado;
15
16    // configurar la GUI
17    public DemoAparienciaVisual()
18    {
19       super( "Demo de apariencia visual" );
20
21       Container contenedor = getContentPane();
```

Figura 14.11 Apariencia visual de una GUI basada en Swing. (Parte 1 de 3.)

http://librosysolucionarios.net

```
22
23          // establecer panel para región NORTH de esquema BorderLayout
24          JPanel panelNorte = new JPanel();
25          panelNorte.setLayout( new GridLayout( 3, 1, 0, 5 ) );
26
27          // establecer etiqueta para panel NORTH
28          etiqueta = new JLabel( "Ésta es la apariencia visual Metal seleccionada",
29             SwingConstants.CENTER );
30          panelNorte.add( etiqueta );
31
32          // establecer botón para panel NORTH
33          boton = new JButton( "JButton" );
34          panelNorte.add( boton );
35
36          // establecer cuadro combinado para panel NORTH
37          cuadroCombinado = new JComboBox( cadenas );
38          panelNorte.add( cuadroCombinado );
39
40          // crear arreglo para botones de opción
41          opcion = new JRadioButton[ cadenas.length ];
42
43          // establecer panel para región SOUTH de esquema BorderLayout
44          JPanel panelSur = new JPanel();
45          panelSur.setLayout( new GridLayout( 1, opcion.length ) );
46
47          // establecer botones de opción para panel SOUTH
48          grupo = new ButtonGroup();
49          ManejadorEventos manejador = new ManejadorEventos();
50
51          for ( int cuenta = 0; cuenta < opcion.length; cuenta++ ) {
52             opcion[ cuenta ] = new JRadioButton( cadenas[ cuenta ] );
53             opcion[ cuenta ].addItemListener( manejador );
54             grupo.add( opcion[ cuenta ] );
55             panelSur.add( opcion[ cuenta ] );
56          }
57
58          // adjuntar paneles NORTH y SOUTH al panel de contenido
59          contenedor.add( panelNorte, BorderLayout.NORTH );
60          contenedor.add( panelSur, BorderLayout.SOUTH );
61
62          // obtener información de la apariencia visual instalada
63          apariencias = UIManager.getInstalledLookAndFeels();
64
65          setSize( 300, 200 );
66          setVisible( true );
67
68          opcion[ 0 ].setSelected( true );
69
70       } // fin del constructor de DemoAparienciaVisual
71
72       // usar UIManager para cambiar la apariencia visual de la GUI
73       private void cambiarLaAparienciaVisual( int valor )
74       {
75          // cambiar la apariencia visual
```



```
76          try {
77             UIManager.setLookAndFeel( apariencias[ valor ].getClassName() );
78             SwingUtilities.updateComponentTreeUI( this );
79          }
80
81          // procesar problemas al cambiar la apariencia visual
82          catch ( Exception excepcion ) {
83             excepcion.printStackTrace();
84          }
85       }
86
87       public static void main( String args[] )
88       {
89          DemoAparienciaVisual aplicacion = new DemoAparienciaVisual();
90          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
91       }
92
93       // clase interna privada para manejar eventos de botones de opción
94       private class ManejadorEventos implements ItemListener {
95
96          // procesar selección de apariencia visual del usuario
97          public void itemStateChanged( ItemEvent evento )
98          {
99             for ( int cuenta = 0; cuenta < opcion.length; cuenta++ )
100
101               if ( opcion[ cuenta ].isSelected() ) {
102                  etiqueta.setText( "Ésta es la apariencia visual " +
103                     cadenas[ cuenta ] + " seleccionada" );
104                  cuadroCombinado.setSelectedIndex( cuenta );
105                  cambiarLaAparienciaVisual( cuenta );
106               }
107         }
108
109      } // fin de la clase interna privada ManejadorEventos
110
111   } // fin de la clase DemoAparienciaVisual
```



Figura 14.11 Apariencia visual de una GUI basada en Swing. (Parte 2 de 3.)

Figura 14.11 Apariencia visual de una GUI basada en Swing. (Parte 3 de 3.)

http://librosysolucionarios.net

Todos los componentes de la GUI y el manejo de eventos de este ejemplo se han descrito anteriormente, por lo que ahora nos concentraremos en el mecanismo para cambiar la apariencia visual. La clase *UIManager* (paquete *javax.swing*) contiene la clase anidada *LookAndFeelInfo* (una clase `public static`) que mantiene la información acerca de una apariencia visual. En la línea 9 se declara un arreglo de tipo `UIManager.LookAndFeelInfo` (observe la sintaxis utilizada para identificar a la clase interna `LookAndFeelInfo`). En la línea 63 se usa el método estático *getInstalledLookAndFeels* de `UIManager` para obtener el arreglo de objetos `UIManager.LookAndFeelInfo` que describe cada una de las apariencias visuales disponibles en su sistema.

Tip de rendimiento 14.1



Cada apariencia visual se representa mediante una clase de Java. El método `getInstalledLookAndFeels` *de* `UIManager` *no carga a cada una de esas clases. En vez de ello, proporciona los nombres de las clases de apariencia visual disponibles, de manera que pueda seleccionarse una de ellas (se supone que una vez, al inicio del programa). Esto reduce la sobrecarga que se genera al cargar clases adicionales que el programa no va a utilizar.*

Nuestro método utilitario `cambiarLaAparienciaVisual` (líneas 73 a 85) es llamado por el manejador de eventos para los objetos `JRadioButtons` que se encuentran en la parte inferior de la interfaz de usuario. El manejador de eventos (declarado en la clase interna `private ManejadorEventos` en las líneas 94 a 109) le pasa un valor entero que representa el elemento en el arreglo `apariencias` que deberá utilizarse para cambiar la apariencia visual. En la línea 77 se utiliza el método estático *setLookAndFeel* de `UIManager` para cambiar la apariencia visual. El método *getClassName* de la clase `UIManager.LookAndFeelInfo` determina el nombre de la clase de apariencia visual que corresponde al objeto `UIManager.LookAndFeelInfo`. Si la clase de apariencia visual no se ha cargado ya, se cargará como parte de la llamada a `setLookAndFeel`. En la línea 78 se utiliza el método estático *updateComponentTreeUI* de *SwingUtilities* (paquete `javax.swing`) para cambiar la apariencia visual de todos los componentes adjuntos a su argumento (la instancia `this` de nuestra clase de aplicación `DemoAparienciaVisual`) a la nueva apariencia visual.

Las líneas 77 y 78 aparecen en un bloque especial de código, conocido como *bloque try*. Este código es parte del *mecanismo de manejo de excepciones*, el cual describiremos detalladamente en el siguiente capítulo. Este código es requerido en caso de que en la línea 77 se trate de cambiar la apariencia visual por una que no exista. En las líneas 82 a 84 se completa el mecanismo de manejo de excepciones con una *cláusula catch* que simplemente procesa este problema (si es que ocurre) imprimiendo un mensaje de error en la línea de comandos. Por ahora, si necesita cambiar la apariencia visual de su GUI, imite el código que se muestra en las líneas 76 a 84.

14.10 `JDesktopPane` y `JInternalFrame`

Muchas de las aplicaciones de hoy en día utilizan una *interfaz de múltiples documentos (MDI)*, es decir, una ventana principal (a la que se le conoce comúnmente como la *ventana padre*) que contiene otras ventanas (a las que se les conoce comúnmente como *ventanas hijas*), para administrar varios *documentos* abiertos que se procesan en paralelo. Por ejemplo, muchos programas de correo electrónico le permiten tener varias ventanas abiertas al mismo tiempo, para que usted pueda componer o leer varios mensajes de correo electrónico de manera simultánea. De manera similar, muchos procesadores de palabras permiten al usuario abrir varios documentos en ventanas separadas, para que el usuario pueda alternar entre los documentos sin tener que cerrar el documento actual para abrir otro. El programa de la figura 14.12 demuestra el uso de las clases *JDesktopPane* y *JInternalFrame* de Swing para implementar interfaces de múltiples documentos. Las ventanas hijas muestran imágenes en este ejemplo.

En las líneas 16 a 23 se crean objetos `JMenuBar`, `JMenu` y `JMenuItem`, se agrega el objeto `JMenuItem` al objeto `JMenu`, se agrega el objeto `JMenu` al objeto `JMenuBar` y se establece el objeto `JMenuBar` para la ventana de aplicación. Cuando el usuario selecciona el objeto `JMenuItem nuevoMarco`, el programa crea y muestra un nuevo objeto `JInternalFrame` que contiene una imagen.

En la línea 26 se asigna la referencia *JDesktopPane* (paquete `javax.swing`), que apunta a la variable `elEscritorio`, a un nuevo objeto `JDesktopPane` que se utilizará para administrar las ventanas hijas `JInternalFrame`. En la línea 27 se agrega el objeto `JDesktopPane` al panel de contenido de la ventana de aplicación. De manera predeterminada, el objeto `JDesktopPane` se agrega al centro del esquema `BorderLayout` del panel de contenido, por lo que el objeto `JDesktopPane` se expande para rellenar toda la ventana de aplicación.

En las líneas 30 a 56 se registra un objeto `ActionListener` para manejar el evento cuando el usuario selecciona el elemento de menú `nuevoMarco`. Cuando ocurre el evento, el método `actionPerformed` (líneas 35 a 52) crea un objeto `JInternalFrame` con las líneas 38 y 39. El constructor de `JInternalFrame` que se utiliza aquí requiere cinco argumentos: una cadena para la barra de título de la ventana interna, un valor `boolean` que indique si el usuario puede reajustar el tamaño del marco interno, un valor `boolean` que indique si el usuario puede cerrar el marco interno, un valor `boolean` que indique si el usuario puede maximizar el marco interno y un valor `boolean` que indique si el usuario puede minimizar el marco interno. Para cada uno de los argumentos `boolean`, un valor de `true` indica que la operación debe permitirse (como se da el caso aquí).

```
1  // Fig. 14.12: PruebaEscritorio.java
2  // Demostración de JDesktopPane.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaEscritorio extends JFrame {
8     private JDesktopPane elEscritorio;
9
10    // configurar la GUI
11    public PruebaEscritorio()
12    {
13       super( "Uso de un objeto JDesktopPane" );
14
15       // crear barra de menús, menú y elemento de menú
16       JMenuBar barra = new JMenuBar();
17       JMenu menuAgregar = new JMenu( "Agregar" );
18       JMenuItem nuevoMarco = new JMenuItem( "Marco interno" );
19
20       menuAgregar.add( nuevoMarco );
21       barra.add( menuAgregar );
22
23       setJMenuBar( barra );
24
25       // establecer escritorio
26       elEscritorio = new JDesktopPane();
27       getContentPane().add( elEscritorio );
28
29       // establecer componente de escucha para elemento de menú nuevoMarco
30       nuevoMarco.addActionListener(
31
32          new ActionListener() {  // clase interna anónima
33
34             // mostrar nueva ventana interna
35             public void actionPerformed( ActionEvent evento ) {
36
37                // crear marco interno
38                JInternalFrame marco = new JInternalFrame(
39                   "Marco interno", true, true, true, true );
40
```

Figura 14.12 Interfaz de múltiples documentos. (Parte 1 de 3.)

http://librosysolucionarios.net

```
                // adjuntar panel al panel de contenido del marco interno
                Container contenedor = marco.getContentPane();
                MiJPanel panel = new MiJPanel();
                contenedor.add( panel, BorderLayout.CENTER );

                // establecer tamaño de marco interno en el tamaño de su contenido
                marco.pack();

                // adjuntar marco interno al escritorio y mostrarlo
                elEscritorio.add( marco );
                marco.setVisible( true );
             }

          } // fin de la clase interna anónima

       ); // fin de la llamada a addActionListener

       setSize( 600, 460 );
       setVisible( true );

    } // fin del constructor

    public static void main( String args[] )
    {
       PruebaEscritorio aplicacion = new PruebaEscritorio();
       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }

} // fin de la clase PruebaEscritorio

// clase para mostrar un objeto ImageIcon en un panel
class MiJPanel extends JPanel {
    private ImageIcon iconoImagen;
    private String[] imagenes = { "floresamarillas.png", "florespurpura.png",
       "floresrojas.png", "floresrojas2.png", "floreslavanda.png" };

    // cargar imagen
    public MiJPanel()
    {
       int numeroAleatorio = ( int ) ( Math.random() * 5 );
       iconoImagen = new ImageIcon( imagenes[ numeroAleatorio ] );
    }

    // mostrar iconoImagen en el panel
    public void paintComponent( Graphics g )
    {
       // llamar al método paintComponent de la superclase
       super.paintComponent( g );

       // mostrar icono
       iconoImagen.paintIcon( this, g, 0, 0 );
    }

    // devolver dimensiones de la imagen
    public Dimension getPreferredSize()
```

Figura 14.12 Interfaz de múltiples documentos. (Parte 2 de 3.)


```
    {
       return new Dimension( iconoImagen.getIconWidth(),
          iconoImagen.getIconHeight() );
    }

} // fin de la clase MiJPanel
```

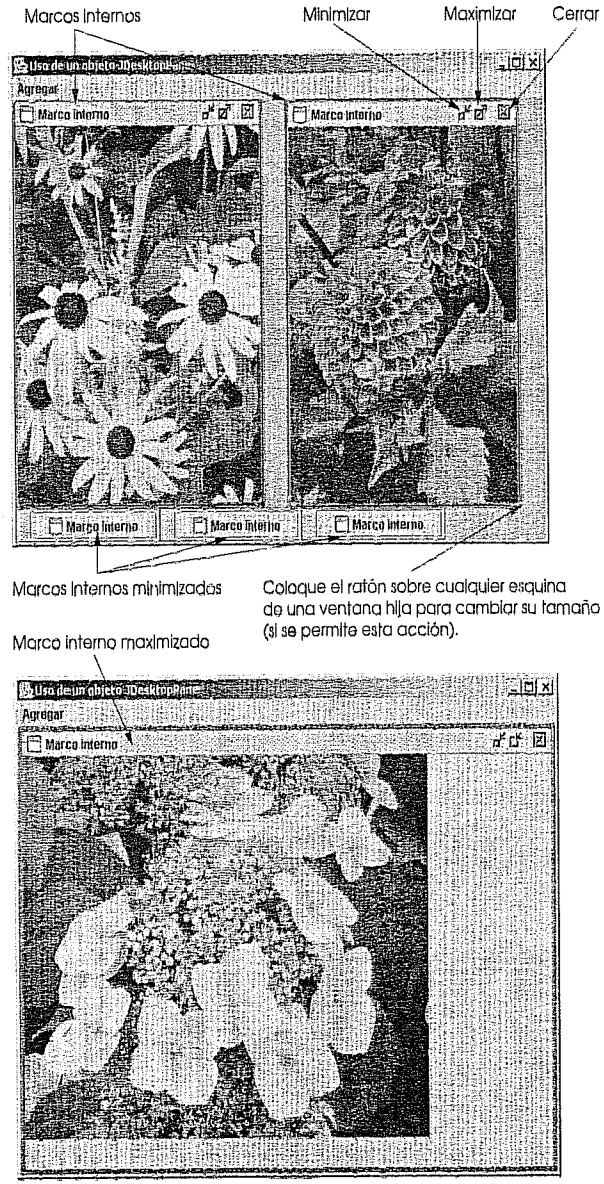


Figura 14.12 Interfaz de múltiples documentos. (Parte 3 de 3.)

http://librosysolucionarios.net

Al igual que con los objetos **JFrame** y **JApplet**, un objeto **JInternalFrame** tiene un panel de contenido, al cual pueden adjuntarse componentes de la GUI. En la línea 42 se obtiene una referencia al panel de contenido del objeto **JInternalFrame**. En la línea 43 se crea una instancia de nuestra clase **MiJPanel** (declarada en las líneas 72 a 101), la cual se agrega al panel de contenido del objeto **JInternalFrame** en la línea 44.

En la línea 47 se utiliza el método ***pack*** de **JInternalFrame** para establecer el tamaño de la ventana hija. El método **pack** utiliza los tamaños preferidos para los componentes en el panel de contenido, para determinar el tamaño de la ventana. La clase **MiJPanel** declara el método **getPreferredSize** (líneas 95 a 99) para especificar el tamaño preferido del panel. En la línea 50 se agrega el objeto **JInternalFrame** al objeto **JDesktopPane**, y en la línea 51 se muestra el objeto **JInternalFrame**.

Las clases **JInternalFrame** y **JDesktopPane** proporcionan muchos métodos para administrar ventanas hijas. Vea la documentación de la API en línea acerca de **JInternalFrame** y **JDesktopPane**, para obtener listas completas de estos métodos:

```
java.sun.com/j2se/1.4.1/docs/api/javax/swing/JInternalFrame.html
java.sun.com/j2se/1.4.1/docs/api/javax/swing/JDesktopPane.html
```

14.11 JTabbedPane

Un objeto ***JTabbedPane*** ordena los componentes de la GUI en capas, en donde sólo una capa está visible en un momento dado. Los usuarios acceden a cada una de las capas mediante una ficha; algo muy parecido a las carpetas en un archivero. Cuando los usuarios hacen clic en una ficha, se muestra la capa apropiada. Las fichas aparecen en la parte superior de manera predeterminada, pero también pueden colocarse a la izquierda, derecha o en la parte inferior del objeto **JTabbedPane**. Puede colocarse cualquier componente en una ficha. Si el componente es un contenedor tal como un panel, puede utilizar cualquier administrador de esquemas para distribuir varios componentes en esa ficha. La clase **JTabbedPane** es una subclase de **JComponent**. El programa de la figura 14.13 crea un panel con tres fichas. Cada ficha muestra uno de los objetos **JPanel**: **panel1**, **panel2** o **panel3**.

```
1  // Fig. 14.13: DemoJTabbedPane.java
2  // Demostración de JTabbedPane.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class DemoJTabbedPane extends JFrame  {
7
8     // configurar la GUI
9     public DemoJTabbedPane()
10    {
11       super( "Demo de JTabbedPane " );
12
13       // crear objeto JTabbedPane
14       JTabbedPane panelConFichas = new JTabbedPane();
15
16       // establecer panel1 y agregarlo al objeto JTabbedPane
17       JLabel etiqueta1 = new JLabel( "panel uno", SwingConstants.CENTER );
18       JPanel panel1 = new JPanel();
19       panel1.add( etiqueta1 );
20       panelConFichas.addTab( "Ficha uno", null, panel1, "Primer panel" );
21
22       // establecer panel2 y agregarlo al objeto JTabbedPane
23       JLabel etiqueta2 = new JLabel( "panel dos", SwingConstants.CENTER );
```

Figura 14.13 Objeto **JTabbedPane** utilizado para organizar componentes de la GUI. (Parte 1 de 2.)


```
24       JPanel panel3 = new JPanel();
25       panel2.setBackground( Color.YELLOW );
26       panel2.add( etiqueta2 );
27       panelConFichas.addTab( "Ficha dos", null, panel2, "Segundo panel" );
28
29       // establecer panel3 y agregarlo al objeto JTabbedPane
30       JLabel etiqueta3 = new JLabel( "panel tres" );
31       JPanel panel3 = new JPanel();
32       panel3.setLayout( new BorderLayout() );
33       panel3.add( new JButton( "Norte" ), BorderLayout.NORTH );
34       panel3.add( new JButton( "Oeste" ), BorderLayout.WEST );
35       panel3.add( new JButton( "Este" ), BorderLayout.EAST );
36       panel3.add( new JButton( "Sur" ), BorderLayout.SOUTH );
37       panel3.add( etiqueta3, BorderLayout.CENTER );
38       panelConFichas.addTab( "Ficha tres", null, panel3, "Tercer panel" );
39
40       // agregar objeto JTabbedPane al contenedor
41       getContentPane().add( panelConFichas );
42
43       setSize( 250, 200 );
44       setVisible( true );
45
46    } // fin del constructor
47
48    public static void main( String args[] )
49    {
50       DemoJTabbedPane demoPanelConFichas = new DemoJTabbedPane();
51       demoPanelConFichas.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
52    }
53
54 } // fin de la clase DemoJTabbedPane
```



Figura 14.13 Objeto **JTabbedPane** utilizado para organizar componentes de la GUI. (Parte 2 de 2.)

El constructor (líneas 9 a 46) crea la GUI. En la línea 14 se crea un objeto **JTabbedPane** vacío con la configuración predeterminada (es decir, fichas en la parte superior). Si las fichas no se ajustan en una línea, pasarán a la siguiente para formar líneas adicionales de fichas. A continuación, el constructor crea los objetos **JPanel** **panel1**, **panel2** y **panel3**, junto con sus componentes de la GUI. A medida que configuramos cada panel, lo agregamos al objeto **panelConFichas** utilizando el método **addTab** de **JTabbedPane** con cuatro argumentos. El primer argumento es una cadena que especifica el título de la ficha. El segundo es una referencia **Icon** que especifica un icono a mostrar en la ficha. Si el objeto **Icon** es una referencia **null**, no se muestra una imagen. El tercer argumento es una referencia **Component** que representa el componente de la GUI a mostrar cuando el usuario hace clic en la ficha. El primer argumento es una cadena que especifica el consejo de información sobre herramientas de la ficha. Por ejemplo, en la línea 20 se agrega el objeto **JPanel** **panel1** al objeto **panelConFichas** con el título **"Ficha uno"** y la información sobre herramientas

http://librosysolucionarios.net

"Primer panel". Los objetos JPanel panel2 y panel3 se agregan a panelConFichas en las líneas 27 y 38. Para ver cada una de las fichas, haga clic en la ficha deseada con el ratón o utilice las teclas de dirección para avanzar por cada una de las fichas.

14.12 Administradores de esquemas: BoxLayout y GridBagLayout

En el capítulo anterior presentamos tres administradores de esquemas: FlowLayout, BorderLayout y GridLayout. En esta sección presentamos dos administradores de esquemas adicionales (los cuales se sintetizan en la figura 14.14). En los siguientes ejemplos, hablaremos sobre estos administradores de esquemas.

Administrador de esquemas BoxLayout

El *administrador de esquemas* BoxLayout ordena los componentes de la GUI en forma horizontal a lo largo del eje *x*, o en forma vertical a lo largo del eje *y* de un contenedor. El programa de la figura 14.15 demuestra el uso del esquema BoxLayout y la clase contenedora Box que utiliza a BoxLayout como su administrador de esquemas predeterminado.

Administrador de esquemas	Descripción
BoxLayout	Un administrador de esquemas que permite ordenar los componentes de la GUI de izquierda a derecha, o de arriba hacia abajo, en un contenedor. La clase Box declara un contenedor con BoxLayout como su administrador de esquemas predeterminado, y proporciona métodos estáticos para crear un objeto Box con un esquema BoxLayout horizontal o vertical.
GridBagLayout	Un administrador de esquemas similar a GridLayout. A diferencia de GridLayout, el tamaño de cada componente puede variar y pueden agregarse componentes en cualquier orden.

Figura 14.14 Administradores de esquemas adicionales.

```
1  // Fig. 14.15: DemoBoxLayout.java
2  // Demostración de BoxLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoBoxLayout extends JFrame {
8
9     // configurar la GUI
10    public DemoBoxLayout()
11    {
12       super( "Demo de BoxLayout" );
13
14       // crear contenedores Box con esquema BoxLayout
15       Box horizontal1 = Box.createHorizontalBox();
16       Box vertical1 = Box.createVerticalBox();
17       Box horizontal2 = Box.createHorizontalBox();
18       Box vertical2 = Box.createVerticalBox();
19
20       final int TAMANIO = 3; // número de botones en cada contenedor Box
21
22       // agregar botones al objeto Box horizontal1
23       for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ )
24          horizontal1.add( new JButton( "Botón " + cuenta ) );
```

Figura 14.15 Administrador de esquemas BoxLayout. (Parte 1 de 3.)

http://librosysolucionarios.net


```
25
26       // crear montante y agregar botones al objeto Box vertical1
27       for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ ) {
28          vertical1.add( Box.createVerticalStrut( 25 ) );
29          vertical1.add( new JButton( "Botón " + cuenta ) );
30       }
31
32       // crear pegamento horizontal y agregar botones al objeto Box horizontal2
33       for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ ) {
34          horizontal2.add( Box.createHorizontalGlue() );
35          horizontal2.add( new JButton( "Botón " + cuenta ) );
36       }
37
38       // crear área rígida y agregar botones al objeto Box vertical2
39       for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ ) {
40          vertical2.add( Box.createRigidArea( new Dimension( 12, 8 ) ) );
41          vertical2.add( new JButton( "Botón " + cuenta ) );
42       }
43
44       // crear pegamento vertical y agregar botones al panel
45       JPanel panel = new JPanel();
46       panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );
47
48       for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ ) {
49          panel.add( Box.createGlue() );
50          panel.add( new JButton( "Botón " + cuenta ) );
51       }
52
53       // crear un objeto JTabbedPane
54       JTabbedPane fichas = new JTabbedPane(
55          JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT );
56
57       // colocar cada contendor en panel con fichas
58       fichas.addTab( "Cuadro horizontal", horizontal1 );
59       fichas.addTab( "Cuadro vertical con montantes", vertical1 );
60       fichas.addTab( "Cuadro horizontal con pegamento", horizontal2 );
61       fichas.addTab( "Cuadro vertical con áreas rígidas", vertical2 );
62       fichas.addTab( "Cuadro vertical con pegamento", panel );
63
64       getContentPane().add( fichas ); // colocar panel con fichas
65                                       // en el panel de contenido
66       setSize( 400, 220 );
67       setVisible( true );
68
69    } // fin del constructor
70
71    public static void main( String args[] )
72    {
73       DemoBoxLayout aplicacion = new DemoBoxLayout();
74       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
75    }
76
77 } // fin de la clase DemoBoxLayout
```

Figura 14.15 Administrador de esquemas BoxLayout. (Parte 2 de 3.)

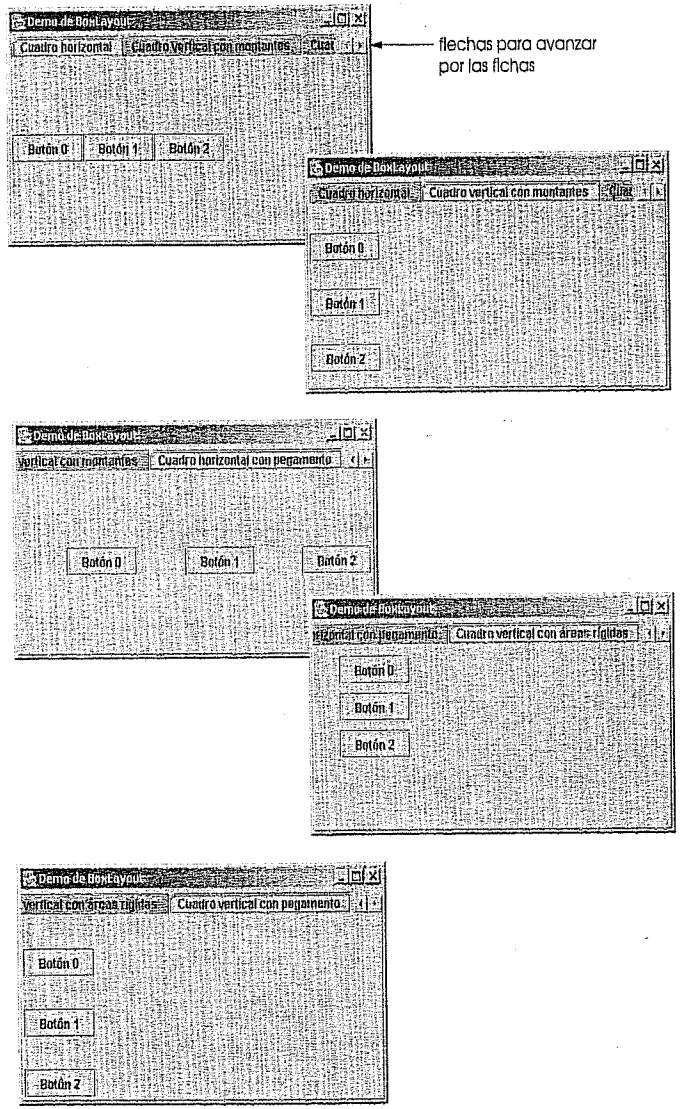


Figura 14.15 Administrador de esquemas `BoxLayout`. (Parte 3 de 3.)

En las líneas 15 a 18 se crean contenedores `Box`. Las referencias `horizontal1` y `horizontal2` se inicializan con el método estático `createHorizontalBox` de `Box`, el cual devuelve un contenedor `Box` con un esquema `BoxLayout` horizontal en el que los componentes de la GUI se ordenan de izquierda a derecha. Las variables `vertical1` y `vertical2` se inicializan con el método estático `createVerticalBox` de `Box`, el cual devuelve referencias a contenedores `Box` con un esquema `BoxLayout` vertical, en el que los componentes de la GUI se ordenan de arriba hacia abajo.

La instrucción `for` de las líneas 23 y 24 agrega tres objetos `JButton` a `horizontal1`. La instrucción `for` de las líneas 27 a 30 agrega tres objetos `JButton` a `vertical1`. Antes de agregar cada botón, en la li-

nea 28 se agrega un *montante vertical* al contenedor, mediante el método estático `createVerticalStrut` de `Box`. Un montante vertical es un componente de la GUI invisible, el cual tiene una altura fija en píxeles y se utiliza para garantizar una cantidad fija de espacio entre los componentes de la GUI. El argumento para el método `createVerticalStrut` determina la altura del montante, en píxeles. Cuando se cambia el tamaño del contenedor, la distancia entre los componentes de la GUI que están separados por montantes no cambia. La clase `Box` también declara el método `createHorizontalStrut` para esquemas `BoxLayout` horizontales.

La instrucción `for` de las líneas 33 a 36 agrega tres objetos `JButton` a `horizontal2`. Antes de agregar cada botón, en la línea 34 se agrega *pegamento horizontal* al contenedor, mediante el método estático `createHorizontalGlue` de `Box`. El pegamento horizontal es un componente de la GUI invisible que puede usarse entre los componentes de la GUI de tamaño fijo para ocupar espacio adicional. Generalmente, el espacio adicional aparece a la derecha del último componente de la GUI horizontal, o debajo del último componente de la GUI vertical, en un esquema `BoxLayout`. El pegamento permite que se agregue espacio adicional entre los componentes de la GUI. Cuando se cambia el tamaño del contenedor, los componentes separados por pegamento conservan el mismo tamaño, pero el pegamento se estira o se contrae para ocupar el espacio entre los demás componentes. La clase `Box` también declara el método `createVerticalGlue` para esquemas `BoxLayout` verticales.

La instrucción `for` de las líneas 39 a 42 agrega tres objetos `JButton` a `vertical2`. Antes de agregar cada botón, en la línea 40 se agrega un *área rígida* al contenedor, mediante el método estático `createRigidArea` de `Box`. Un área rígida es un componente de la GUI invisible, el cual siempre tiene una anchura y altura fijas, en píxeles. El argumento para el método `createRigidArea` es un objeto `Dimension` que especifica la anchura y la altura del área rígida.

En las líneas 45 a 46 se crea un objeto `JPanel` y se establece su esquema en `BoxLayout` de la manera convencional, utilizando el método `setLayout` de `Container`. El constructor de `BoxLayout` recibe una referencia al contenedor para el que está controlando el esquema, y una constante que indica si el esquema es horizontal (`BoxLayout.X_AXIS`) o vertical (`BoxLayout.Y_AXIS`).

La instrucción `for` de las líneas 48 a 51 agrega tres objetos `JButton` al objeto `panel`. Antes de agregar cada botón, en la línea 49 se agrega un componente pegamento al contenedor, mediante el método estático `createGlue` de `Box`. Este componente se expande o se contrae con base en el tamaño del objeto `Box`.

En las líneas 54 a 55 se crea un objeto `JTabbedPane` para mostrar los cinco contenedores en este programa. Los argumentos para el constructor indican que las fichas deben aparecer en la parte superior del objeto `JTabbedPane`, y que las fichas deben desplazarse si hay demasiadas como para que puedan ajustarse en una sola línea.

Los contenedores `Box` y el objeto `JPanel` se adjuntan al objeto `JTabbedPane` en las líneas 58 a 62. Ahora pruebe a ejecutar la aplicación. Cuando aparezca la ventana, cambie su tamaño para ver cómo afectan los componentes pegamento, montante y área rígida en la distribución de cada ficha.

Administrador de esquemas `GridBagLayout`

El más complejo y poderoso de los administradores de esquemas predefinidos es `GridBagLayout`. Este esquema es similar a `GridLayout`, ya que también ordena los componentes en una cuadrícula. Sin embargo, el esquema `GridBagLayout` es más flexible. Los componentes pueden variar en tamaño (es decir, pueden ocupar varias filas y columnas) y pueden agregarse en cualquier orden.

El primer paso para utilizar `GridBagLayout` es determinar la apariencia de la GUI. Este paso no implica ningún tipo de programación; todo lo que se necesita es un pedazo de papel. Primero, dibuje la GUI. Después, dibuje una cuadrícula sobre la GUI, dividiendo los componentes en filas y columnas. Los números iniciales de fila y columna deben ser 0, de manera que el esquema `GridBagLayout` pueda colocar apropiadamente los componentes en la cuadrícula. Los números de fila y columna se utilizarán para colocar cada uno de los componentes en una posición exacta en la cuadrícula. En la figura 14.16 se demuestra cómo dibujar las líneas para las filas y columnas sobre una GUI.

Un objeto `GridBagConstraints` describe cómo colocar un componente en un esquema `GridBagLayout`. Varios campos de `GridBagConstraints` se sintetizan en la figura 14.17.

Las variables `gridx` y `gridy` especifican que la fila y columna pueden ser "relativas", así como la posición que tendrá la esquina superior izquierda del componente en la cuadrícula. La variable `gridx` corresponde

http://librosysolucionarios.net

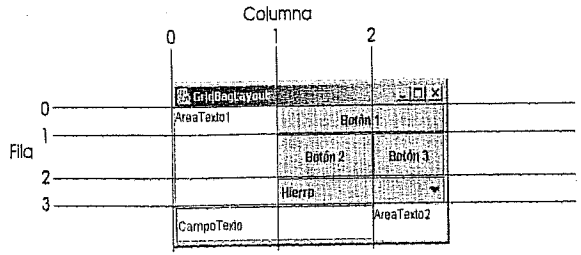


Figura 14.16 Diseño de una GUI que utiliza el esquema `GridBagLayout`.

a la columna, y la variable `gridy` corresponde a la fila. En la figura 14.16, el objeto `JComboBox` (que muestra la cadena "`Hierro`") tiene un valor de `gridx` de 1 y un valor de `gridy` de 2.

La variable ***`gridwidth`*** especifica el número de columnas que ocupa un componente. El objeto `JComboBox` ocupa dos columnas. La variable ***`gridheight`*** especifica el número de filas que ocupa un componente. El objeto `JTextArea` del lado izquierdo de la ventana ocupa tres filas.

La variable ***`weightx`*** especifica cómo distribuir el espacio horizontal adicional en las ranuras de cuadrícula de un esquema `GridBagLayout`, cuando se ajusta el tamaño del contenedor. Un valor de cero indica que la ranura de la cuadrícula no crecerá horizontalmente por sí sola. No obstante, si el componente abarca una columna que contenga un componente con un valor de `weightx` distinto de cero, el componente con valor de `weightx` de cero crecerá horizontalmente en la misma proporción que el (los) otro(s) componente(s) en la misma columna. Esto se debe a que cada componente debe mantenerse en las mismas fila y columna en las que se colocó originalmente.

La variable ***`weighty`*** especifica cómo distribuir el espacio vertical adicional en las ranuras de cuadrícula de un esquema `GridBagLayout`, cuando se ajusta el tamaño del contenedor. Un valor de cero indica que la ranura de la cuadrícula no crecerá verticalmente por sí sola. No obstante, si el componente abarca una colum-

Campo de `GridBagConstraints`	Descripción
`fill`	Ajusta el tamaño del componente en la dirección especificada (`NONE`, `HORIZONTAL`, `VERTICAL`, `BOTH`) cuando el área en pantalla es más grande que el componente.
`gridx`	La columna en la que se colocará el componente.
`gridy`	La fila en la que se colocará el componente.
`gridwidth`	El número de columnas que ocupa el componente.
`gridheight`	El número de filas que ocupa el componente.
`weightx`	La porción de espacio adicional que se asignará horizontalmente. La ranura de la cuadrícula puede hacerse más ancha cuando hay espacio adicional disponible.
`weighty`	La porción de espacio adicional que se asignará verticalmente. La ranura de la cuadrícula puede hacerse más alta cuando hay espacio adicional disponible.

Figura 14.17 Campos de `GridBagConstraints`.

na que contenga un componente con un valor de `weighty` distinto de cero, el componente con valor de `weighty` de cero crecerá verticalmente en la misma proporción que el (los) otro(s) componente(s) en la misma columna.

En la figura 14.16, los efectos de `weighty` y `weightx` no podrán verse fácilmente sino hasta que se ajuste el tamaño del contenedor y haya más espacio adicional disponible. Los componentes con valores mayores para `weightx` y `weighty` ocupan más espacio adicional que los componentes con valores menores.

Los componentes deben recibir valores positivos distintos de cero para `weightx` y `weighty`; de lo contrario, se "amontonarán" al centro del contenedor. En la figura 14.18 se muestra la GUI de la figura 14.16, con un valor de cero para `weightx` y `weighty`.

El campo `fill` de `GridBagConstraints` define la manera en que el componente crece, si el área en la que puede mostrarse es mayor que él. A la variable `fill` se le puede asignar una de las siguientes constantes de `GridBagConstraints`: ***`NONE`***, ***`VERTICAL`***, ***`HORIZONTAL`*** o ***`BOTH`***. El valor predeterminado es `NONE`, el cual indica que el componente no crecerá en ninguna dirección. `VERTICAL` indica que el componente crecerá verticalmente. `HORIZONTAL` indica que el componente crecerá horizontalmente. `BOTH` indica que el componente crecerá en ambas direcciones.

El campo ***`anchor`*** de `GridBagConstraints` especifica la posición relativa del componente, en un área que éste no ocupa por completo. A la variable `anchor` se le puede asignar una de las siguientes constantes de `GridBagConstraints`: ***`NORTH`***, ***`NORTHEAST`***, ***`EAST`***, ***`SOUTHEAST`***, ***`SOUTH`***, ***`SOUTHWEST`***, ***`WEST`***, ***`NORTHWEST`*** o ***`CENTER`***. El valor predeterminado es `CENTER`.

El programa de la figura 14.19 utiliza el administrador de esquemas `GridBagLayout` para ordenar los componentes que se encuentran en la GUI de la figura 14.16. Este programa no hace nada más que demostrar cómo utilizar el esquema `GridBagLayout`.

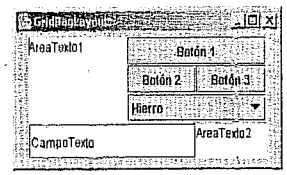


Figura 14.18 `GridBagLayout` con valores de cero para `weightx` y `weighty`.

```
1  // Fig. 14.19: DemoGridBag.java
2  // Demostración de GridBagLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoGridBag extends JFrame {
8     private Container contenedor;
9     private GridBagLayout esquema;
10    private GridBagConstraints restricciones;
11
12    // configurar la GUI
13    public DemoGridBag()
14    {
15       super( "GridBagLayout" );
16
17       contenedor = getContentPane();
18       esquema = new GridBagLayout();
```

Figura 14.19 Administrador de esquemas `GridBagLayout`. (Parte 1 de 3.)

http://librosysolucionarios.net

```
19          contenedor.setLayout( esquema );
20
21          // instanciar restricciones de GridBagLayout
22          restricciones = new GridBagConstraints();
23
24          // crear componentes de la GUI
25          JTextArea areaTexto1 = new JTextArea( "AreaTexto1", 5, 10 );
26          JTextArea areaTexto2 = new JTextArea( "AreaTexto2", 2, 2 );
27
28          String nombres[] = { "Hierro", "Acero", "Cobre" };
29          JComboBox cuadroCombinado = new JComboBox( nombres );
30
31          JTextField campoTexto = new JTextField( "CampoTexto" );
32          JButton boton1 = new JButton( "Botón 1" );
33          JButton boton2 = new JButton( "Botón 2" );
34          JButton boton3 = new JButton( "Botón 3" );
35
36          // weightx y weighty para areaTexto1 son 0: el valor predeterminado
37          // anchor para todos los componentes es CENTER: el valor predeterminado
38          restricciones.fill = GridBagConstraints.BOTH;
39          agregarComponente( areaTexto1, 0, 0, 1, 3 );
40
41          // weightx y weighty para boton1 son 0: el valor predeterminado
42          restricciones.fill = GridBagConstraints.HORIZONTAL;
43          agregarComponente( boton1, 0, 1, 2, 1 );
44
45          // weightx y weighty para cuadroCombinado son 0: el valor predeterminado
46          // fill es HORIZONTAL
47          agregarComponente( cuadroCombinado, 2, 1, 2, 1 );
48
49          //  boton2
50          restricciones.weightx = 1000;   // puede hacerse más ancho
51          restricciones.weighty = 1;      // puede hacerse más largo
52          restricciones.fill = GridBagConstraints.BOTH;
53          agregarComponente( boton2, 1, 1, 1, 1 );
54
55          // fill es BOTH para boton3
56          restricciones.weightx = 0;
57          restricciones.weighty = 0;
58          agregarComponente( boton3, 1, 2, 1, 1 );
59
60          // weightx y weighty para campoTexto son 0, fill es BOTH
61          agregarComponente( campoTexto, 3, 0, 2, 1 );
62
63          // weightx y weighty para areaTexto2 son 0, fill es BOTH
64          agregarComponente( areaTexto2, 3, 2, 1, 1 );
65
66          setSize( 300, 150 );
67          setVisible( true );
68
69       }// fin del constructor de DemoGridBag
70
71       // método para establecer restricciones
72       private void agregarComponente( Component componente,
73          int fila, int columna, int anchura, int altura )
```

Figura 14.19 Administrador de esquemas **GridBagLayout**. (Parte 2 de 3.)


```
74       {
75          // establecer gridx y gridy
76          restricciones.gridx = columna;
77          restricciones.gridy = fila;
78
79          // establecer gridwidth y gridheight
80          restricciones.gridwidth = anchura;
81          restricciones.gridheight = altura;
82
83          // establecer restricciones y agregar componente
84          esquema.setConstraints( componente, restricciones );
85          contenedor.add( componente );
86       }
87
88       public static void main( String args[] )
89       {
90          DemoGridBag aplicacion = new DemoGridBag();
91          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
92       }
93
94    } // fin de la clase DemoGridBag
```

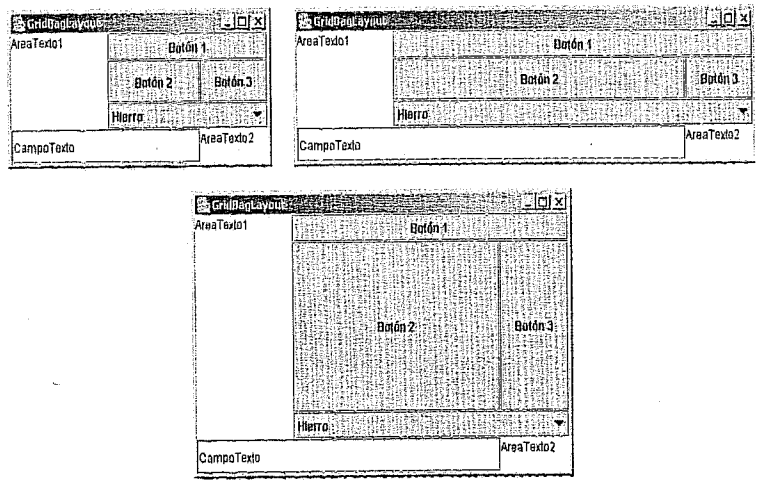


Figura 14.19 Administrador de esquemas **GridBagLayout**. (Parte 3 de 3.)

La GUI consiste de tres objetos **JButton**, dos objetos **JTextArea**, un objeto **JComboBox** y un objeto **JTextField**. El administrador de esquemas para el panel de contenido es **GridBagLayout**. En las líneas 18 a 19 se crea el objeto **GridBagLayout** y se establece el administrador de esquemas para el panel de contenido en **esquema**. En la línea 22 se crea el objeto **GridBagConstraints** utilizado para determinar la ubicación y el tamaño de cada uno de los componentes en la cuadrícula. En las líneas 25 a 34 se crea cada uno de los componentes de la GUI que se agregarán al panel de contenido.

En las líneas 38 a 39 se configura el objeto **JTextArea areaTexto1** y se agrega al panel de contenido. Los valores para **weightx** y **weighty** no se especifican en **restricciones**, por lo que cada una de ellas tiene el valor de cero, de manera predeterminada. Por lo tanto, el objeto **JTextArea** no se cambiará de tamaño a sí mismo, incluso si hay espacio disponible. Sin embargo, el objeto **JTextArea** abarca varias filas por lo que el tamaño vertical está sujeto a los valores de **weightx** y **weighty** de los objetos **JButton**

http://librosysolucionarios.net

boton2 y **boton3**. Cuando se cambie el tamaño de uno de los objetos **boton2** o **boton3** en forma vertical, con base en sus valores de **weightx** y **weighty**, también se cambiará el tamaño del objeto **JTextArea**.

En la línea 38 se establece la variable **fill** de **restricciones** en **GridBagConstraints.BOTH**, lo cual hará que el objeto **JTextArea** siempre llene toda su área asignada en la cuadrícula. No se especifica un valor para **anchor** en **restricciones**, por lo que se utiliza el valor predeterminado de **CENTER**. Como no utilizamos la variable **anchor** en este programa, todos los componentes utilizarán su valor predeterminado. En la línea 39 se hace una llamada a nuestro método utilitario ***agregarComponente*** (declarado en las líneas 72 a 86). El objeto **JTextArea**, la fila, la columna, el número de columnas y el número de filas a abarcar se pasan como argumentos.

Los parámetros del método **agregarComponente** son **componente** (una referencia **Component**) y los enteros **fila**, **columna**, **anchura** y **altura**. En las líneas 76 y 77 se establecen las variables **gridx** y **gridy** de **GridBagConstraints**. A la variable **gridx** se le asigna la columna en la que se va a colocar el objeto **Component**, y a la variable **gridy** se le asigna la fila en la que se va a colocar este objeto. En las líneas 80 y 81 se establecen las variables **gridwidth** y **gridheight** de **GridBagConstraints**. La variable **gridwidth** especifica el número de columnas que abarcará el objeto **Component** en la cuadrícula, y la variable **gridheight** especifica el número de filas que abarcará el objeto **Component**. En la línea 84 se establece el objeto **GridBagConstraints** para un componente en el esquema **GridBagLayout**. El método ***setConstraints*** de la clase **GridBagLayout** toma un argumento **Component** y un argumento **GridBagConstraints**. En la línea 85 se agrega el componente al panel de contenido.

El objeto **JButton boton1** es el siguiente componente que se agrega (líneas 42 y 43). Los valores de **weightx** y **weighty** siguen siendo cero. La variable **fill** se establece en **HORIZONTAL**; el componente siempre ocupará toda su área en dirección horizontal. La dirección vertical no se ocupará toda.

El valor de **weighty** es cero, por lo que el botón sólo será más alto si otro componente en la misma fila tiene un valor de **weighty** distinto de cero. El objeto **JButton boton1** se encuentra en la fila 0, columna 1. Se ocupan una fila y dos columnas.

El objeto **JComboBox cuadroCombinado** es el siguiente componente que se agrega (línea 47). Los valores de **weightx** y **weighty** son cero, y la variable **fill** se establece en **HORIZONTAL**. El botón **JComboBox** crecerá solamente en dirección horizontal. Observe que las variables **weightx**, **weighty** y **fill** permanecerán establecidas en **restricciones** hasta que se modifiquen. El botón **JComboBox** se coloca en la fila 2, columna 1. Se ocupan una fila y dos columnas.

El objeto **JButton boton2** es el siguiente componente que se agrega (líneas 50 a 53). Obtiene un valor de **weightx** de **1000** y un valor de **weighty** de **1**. El área ocupada por el botón es capaz de crecer en las direcciones vertical y horizontal. La variable **fill** se establece en **BOTH**, lo cual especifica que el botón siempre ocupará toda el área. Cuando se ajuste el tamaño de la ventana, **boton2** crecerá. El botón se coloca en la fila 1, columna 1. Se ocupan una fila y una columna.

El objeto **JButton boton3** se agrega a continuación (líneas 56 a 58). Los valores de **weightx** y **weighty** son cero, y la variable **fill** se establece en **BOTH**. El objeto **JButton boton3** crecerá si se ajusta el tamaño de la ventana; se verá afectado por los valores de **weightx** y **weighty** de **boton2**. Observe que el valor de **weightx** para **boton2** es mucho mayor que el de **boton3**. Cuando ocurra el cambio de tamaño, **boton2** ocupará un mayor porcentaje del nuevo espacio. El botón se coloca en la fila 1, columna 2. Se ocupan una fila y una columna.

Tanto el objeto **JTextField campoTexto** (línea 61) como el objeto **JTextArea areaTexto2** (línea 64) tienen un valor de **weightx** de 0 y un valor de **weighty** de 0. El valor de **fill** es **BOTH**. El objeto **JTextField** se coloca en la fila 3, columna 0, y el objeto **JTextArea** se coloca en la fila 3, columna 2. El objeto **JTextField** ocupa una fila y dos columnas. El objeto **JTextArea** ocupa una fila y una columna.

Cuando ejecute esta aplicación, pruebe a cambiar el tamaño de la ventana para ver cómo las restricciones de cada componente de la GUI afectan su posición y tamaño en la ventana.

Las constantes RELATIVE y REMAINDER de GridBagConstraints

Hay una variación de **GridBagLayout** que no utiliza a las variables **gridx** y **gridy**. En vez de estas variables, se utilizan las constantes ***RELATIVE*** y ***REMAINDER*** de **GridBagConstraints**. Especifica que el penúltimo componente de cierta fila deberá colocarse a la derecha del componente anterior en esa


```
1  // Fig. 14.20: DemoGridBag2.java
2  // Demostración de las constantes de GridBagLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DemoGridBag2 extends JFrame {
8     private GridBagLayout esquema;
9     private GridBagConstraints restricciones;
10    private Container contenedor;
11
12    // configurar la GUI
13    public DemoGridBag2()
14    {
15       super( "GridBagLayout" );
16
17       contenedor = getContentPane();
18       esquema = new GridBagLayout();
19       contenedor.setLayout( esquema );
20
21       // instanciar restricciones de GridBagLayout
22       restricciones = new GridBagConstraints();
23
24       // crear componentes de la GUI
25       String metales[] = { "Cobre", "Aluminio", "Plata" };
26       JComboBox cuadroCombinado = new JComboBox( metales );
27
28       JTextField campoTexto = new JTextField( "CampoTexto" );
29
30       String tiposLetra[] = { "Serif", "Monospaced" };
31       JList lista = new JList( tiposLetra );
32
33       String nombres[] = { "cero", "uno", "dos", "tres", "cuatro" };
34       JButton botones[] = new JButton[ nombres.length ];
35
36       for ( int cuenta = 0; cuenta < botones.length; cuenta++ )
37          botones[ cuenta ] = new JButton( nombres[ cuenta ] );
38
39       // definir restricciones del componente de la GUI campoTexto
40       restricciones.weightx = 1;
41       restricciones.weighty = 1;
42       restricciones.fill = GridBagConstraints.BOTH;
43       restricciones.gridwidth = GridBagConstraints.REMAINDER;
44       agregarComponente( campoTexto );
45
46       // botones[0] -- weightx y weighty son 1: fill es BOTH
47       restricciones.gridwidth = 1;
48       agregarComponente( botones[ 0 ] );
49
50       // botones[1] -- weightx y weighty son 1: fill es BOTH
51       restricciones.gridwidth = GridBagConstraints.RELATIVE;
52       agregarComponente( botones[ 1 ] );
53
```

http://librosysolucionarios.net

Figura 14.20 Las constantes **RELATIVE** y **REMAINDER** de **GridBagConstraints**. (Parte 1 de 2.)

```
54          // botones[2] -- weightx y weighty son 1: fill es BOTH
55          restricciones.gridwidth = GridBagConstraints.REMAINDER;
56          agregarComponente( botones[ 2 ] );
57
58          // cuadroCombinado -- weightx es 1: fill es BOTH
59          restricciones.weighty = 0;
60          restricciones.gridwidth = GridBagConstraints.REMAINDER;
61          agregarComponente( cuadroCombinado );
62
63          // botones[3] -- weightx es 1: fill es BOTH
64          restricciones.weighty = 1;
65          restricciones.gridwidth = GridBagConstraints.REMAINDER;
66          agregarComponente( botones[ 3 ] );
67
68          // botones[4] -- weightx y weighty son 1: fill es BOTH
69          restricciones.gridwidth = GridBagConstraints.RELATIVE;
70          agregarComponente( botones[ 4 ] );
71
72          // lista -- weightx y weighty son 1: fill es BOTH
73          restricciones.gridwidth = GridBagConstraints.REMAINDER;
74          agregarComponente( lista );
75
76          setSize( 300, 200 );
77          setVisible( true );
78
79       } // fin del constructor
80
81       // agregar un componente al contenedor
82       private void agregarComponente( Component componente )
83       {
84          esquema.setConstraints( componente, restricciones );
85          contenedor.add( componente );      // agregar componente
86       }
87
88       public static void main( String args[] )
89       {
90          DemoGridBag2 aplicacion = new DemoGridBag2();
91          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
92       }
93
94    } // fin de la clase DemoGridBag2
```

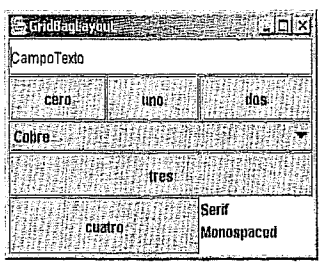


Figura 14.20 Las constantes `RELATIVE` y `REMAINDER` de `GridBagConstraints`. (Parte 2 de 2.)

http://librosysolucionarios.net

fila. `REMAINDER` especifica que un componente es el último en una fila. Cualquier componente que no sea el penúltimo o el último en una fila, deberá especificar valores para las variables `gridwidth` y `gridheight` de `GridbagConstraints`. La clase `DemoGridBag` de la figura 14.20 ordena los componentes de un esquema `GridBagLayout`, utilizando estas constantes.

En las líneas 18 y 19 se crea el objeto `GridBagLayout` y se utiliza para establecer el administrador de esquemas del panel de contenido. Los componentes que se colocan en el esquema `GridBagLayout` se crean en las líneas 25 a 37. Estos componentes son cinco objetos `JButton`, un objeto `JTextField`, un objeto `JList` y un objeto `JComboBox`.

El objeto `JTextField` se agrega primero (líneas 40 a 44). Los valores de `weightx` y `weighty` se establecen en 1, y la variable `fill` se establece en `BOTH`. En la línea 43 se especifica que el objeto `JTextField` es el último componente de la línea. El objeto `JTextField` se agrega al panel de contenido mediante una llamada a nuestro método utilitario `agregarComponente` (declarado en las líneas 82 a 86). El método `agregarComponente` toma un argumento `Component` y utiliza el método `setConstraints` de `GridBagLayout` para establecer las restricciones de este objeto `Component`. El método `add` adjunta el componente al panel de contenido.

El objeto `JButton botones[ 0 ]` (líneas 47 y 48) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Como `botones[ 0 ]` no es uno de los últimos componentes de la fila, recibe un valor de `gridwidth` de 1, de manera que ocupe solamente una columna. El objeto `JButton` se agrega al panel de contenido mediante una llamada al método utilitario `agregarComponente`.

El objeto `JButton botones[ 1 ]` (líneas 51 y 52) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. En la línea 51 se especifica que el objeto `JButton` se va a colocar de manera relativa al componente anterior. El objeto `JButton` se agrega al objeto `JFrame` mediante una llamada a `agregarComponente`.

El objeto `JButton botones[ 2 ]` (líneas 55 y 56) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Este objeto `JButton` es el último componente de la línea, por lo que se utiliza `REMAINDER`. El objeto `JButton` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JComboBox` (líneas 59 a 61) tiene un valor de `weightx` de 1 y un valor de `weighty` de 0. El objeto `JComboBox` no crecerá en dirección vertical. Este objeto `JComboBox` es el último componente de la línea, por lo que se utiliza `REMAINDER`. El objeto `JComboBox` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JButton botones[ 3 ]` (líneas 64 a 66) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Este objeto `JButton` es el último componente de la línea, por lo que se utiliza `REMAINDER`. El objeto `JButton` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JButton botones[ 4 ]` (líneas 69 y 70) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Este objeto `JButton` es el penúltimo componente de la línea, por lo que se utiliza `RELATIVE`. El objeto `JButton` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JList` (líneas 73 y 74) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. El objeto `JList` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

14.13 (Ejemplo práctico opcional) Acerca de los objetos: Modelo-vista-controlador

Los *patrones de diseño* describen estrategias comprobadas para crear sistemas confiables de software orientado a objetos. Nuestro ejemplo práctico se adhiere a la arquitectura MVC (*Modelo-vista-controlador*), la cual utiliza varios patrones de diseño.[2] MVC divide a las responsabilidades del sistema en tres partes:

2. Para aquellos lectores que busquen estudiar de manera más detallada los patrones de diseño y la arquitectura MVC, les recomendamos que lean nuestro material "Descubrimiento de los patrones de diseño" en las secciones 1.16, 9.24, 13.18, 15.13, 17.11 y 21.12.

1. el *modelo*, que mantiene los datos y la lógica del programa;
2. la *vista* (o vistas), que proporciona una presentación visual del modelo y
3. el *controlador*, que procesa la entrada del usuario y hace modificaciones al modelo.

Mediante el uso del controlador, el usuario modifica los datos en el modelo MVC. Después, el MVC informa a la vista acerca de la modificación de los datos. La vista cambia su presentación visual para reflejar los cambios en el modelo MVC.

Por ejemplo, en nuestra simulación el usuario agrega una `Persona` al modelo MVC oprimiendo el objeto `JButton` **Primer Piso** o **Segundo Piso** en el controlador (vea las figuras 2.22 a 2.24). Después, el modelo MVC informa a las vista que se creó la `Persona`. La vista, en respuesta a esta notificación, muestra a una `Persona` en un `Piso`. El modelo MVC desconoce la manera en que la vista muestra a la `Persona`, y la vista desconoce la manera o por qué el modelo MVC creó a la `Persona`.

La arquitectura MVC ayuda a crear sistemas confiables y fáciles de modificar. Si deseamos una salida basada en texto, en vez de una salida gráfica para la simulación del elevador, podemos crear una vista alternativa para producir salida basada en texto, sin tener que alterar el modelo o el controlador de la arquitectura MVC. También podríamos proporcionar una vista tridimensional que utilice una perspectiva en primera persona, para permitir al usuario "formar parte" de la simulación; dichas vistas se utilizan comúnmente en sistemas basados en realidad virtual.

Simulación del elevador con Modelo-vista-controlador

Ahora aplicaremos la arquitectura MVC a nuestra simulación del elevador. Cada diagrama de UML que hemos proporcionado hasta este punto, modela una porción del modelo MVC de nuestro sistema de elevador. En la figura 14.21 proporcionamos un modelo de UML de "mayor nivel" para la simulación. La clase `EjemploPracticoElevador` (una subclase de `JFrame`) agrega una instancia de cada una de las clases `SimulacionElevador`, `VistaElevador` y `ControladorElevador` para crear la aplicación `EjemploPracticoElevador`. En UML, un rectángulo con la esquina superior derecha "doblada" representa a una *nota*. En este caso, cada nota apunta a una clase específica (mediante una línea punteada) para describir la función de esa clase en el sistema. Las clases `SimulacionElevador`, `VistaElevador` y `ControladorElevador` encapsulan a todos los objetos que componen las porciones del modelo, vista y controlador de MVC que corresponden a nuestra simulación, respectivamente.

La clase `SimulacionElevador` es una agregación de varias clases (por ejemplo, `ConductoElevador`, `Elevador` y `Piso`). Para ahorrar espacio, no modelamos esta agregación en la figura 14.21. La clase `VistaElevador` también es una agregación de varias clases; expandiremos el modelo de UML de `VistaElevador` en la sección 19.7 para mostrar estas clases adicionales. La clase `ControladorElevador` representa el controlador de la simulación. Observe que la clase `VistaElevador` implementa a la interfaz `EscuchaSimulacionElevador`, la cual permite a `VistaElevador` recibir eventos del modelo MVC.

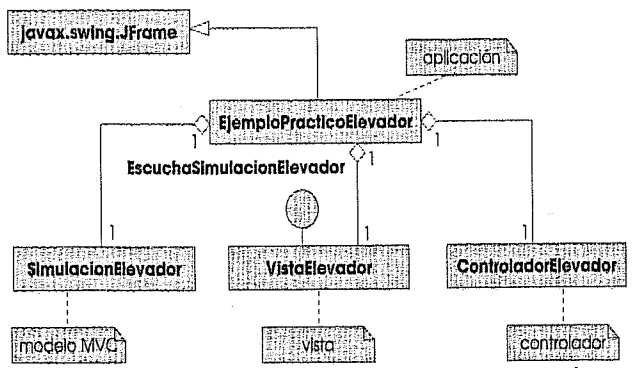


Figura 14.21 Diagrama de clases de la simulación del elevador.

Observación de ingeniería de software 14.1

Siempre que sea apropiado, debe particionar un modelo de UML en varios diagramas más pequeños, de manera que cada diagrama represente a un subsistema único.

La clase `EjemploPracticoElevador` no contiene más atributos que sus referencias a un objeto `SimulacionElevador`, un objeto `VistaElevador` y un objeto `ControladorElevador`. El único comportamiento para la clase `EjemploPracticoElevador` es iniciar el programa; por lo tanto, en Java, dicha la clase contiene un método `main` estático que instancia a un objeto `EjemploPracticoElevador`, el cual a su vez instancia a los objetos `SimulacionElevador`, `VistaElevador` y `ControladorElevador`. Implementaremos la clase `EjemploPracticoElevador` en Java más adelante en esta sección.

Artefactos

La figura 14.21 nos ayuda a diseñar otro aspecto de nuestro sistema: los *artefactos*. En la figura 14.22 se modelan las "piezas" (llamadas *artefactos*) que el sistema necesita para realizar sus tareas. Estas piezas incluyen programas binarios ejecutables, archivos `.class` compilados, archivos de código fuente `.java`, imágenes, paquetes, recursos, etcétera.

En la figura 14.22, cada rectángulo modela a un artefacto. Nuestro sistema contiene cinco artefactos: `EjemploPracticoElevador.class`, `EjemploPracticoElevador.java`, `SimulacionElevador.java`, `VistaElevador.java` y `ControladorElevador.java`.

En la figura 14.22, los gráficos que se asemejan a las carpetas (cuadros con fichas en su esquina superior izquierda) representan *paquetes* en UML (los cuales no deben confundirse con los paquetes de Java). Podemos agrupar a las clases, objetos, artefactos, casos de uso, etcétera, en paquetes. En este diagrama (y en el resto de nuestro ejemplo práctico), los paquetes de UML corresponden a paquetes de Java (los cuales se presentaron en la sección 8.13). En nuestra discusión, utilizamos el tipo de letra Courier en negrita y minúscula para los nombres de los paquetes. Los paquetes en nuestro sistema son `modelo` (contiene las clases del modelo MVC), `vista` (contiene las clases relacionadas con la vista) y `controlador` (contiene las clases relacionadas con

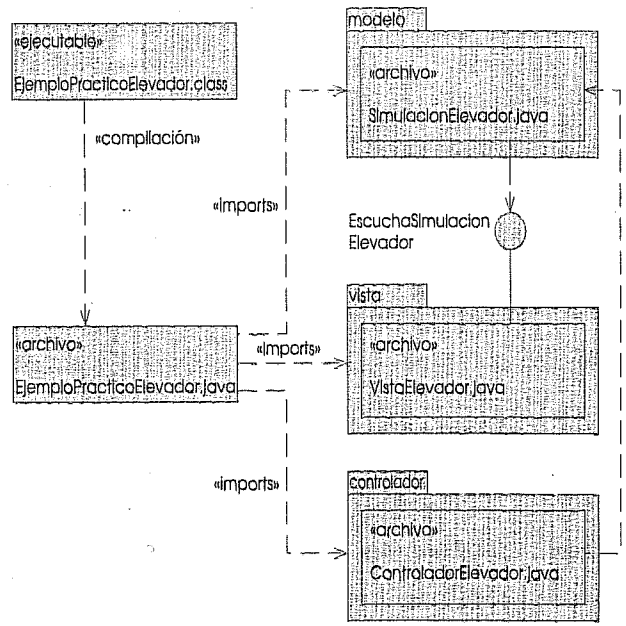


Figura 14.22 Artefactos de la simulación del elevador.

http://librosysolucionarios.net

el controlador). El artefacto `EjemploPracticoElevador.java` contiene una instancia de cada uno de todos los artefactos en estos paquetes. Actualmente, cada paquete contiene sólo un artefacto: un archivo `.java`. El paquete `modelo` contiene a `SimulacionElevador.java`, el paquete `vista` contiene a `VistaElevador.java` y el paquete `controlador` contiene a `ControladorElevador.java`. En los apéndices agregaremos artefactos a cada uno de estos paquetes, cuando implementemos cada una de las clases en nuestro modelo de UML como un artefacto (archivo `.java`).

Cada flecha punteada en la figura 14.22 indica una *dependencia* entre artefactos; la dirección de la flecha indica la relación "depende de". Una dependencia describe la relación entre artefactos, en donde los cambios en un artefacto afectan a otro artefacto. Por ejemplo, el artefacto `EjemploPracticoElevador.class` depende del artefacto `EjemploPracticoElevador.java`, ya que un cambio en este último afecta al primero cuando se compila `EjemploPracticoElevador.java`. El objeto `ControladorElevador` contiene una referencia al objeto `SimulacionElevador` (para colocar objetos `Persona` en objetos `Piso`). Por lo tanto, `ControladorElevador.java` depende de `SimulacionElevador.java`.

De acuerdo con la figura 14.22, `SimulacionElevador.java` y `VistaElevador.java` no dependen uno del otro; se comunican a través de la interfaz `EscuchaSimulacionElevador`, la cual implementa a todas las interfaces en la simulación. `VistaElevador.java` realiza a la interfaz `EscuchaSimulacionElevador`, y `SimulacionElevador.java` depende de la interfaz `EscuchaSimulacionElevador`.

La figura 14.22 contiene varios estereotipos. En la sección 12.9 mencionamos el estereotipo «InterfazJava». El estereotipo «compilación» describe la dependencia entre `EjemploPracticoElevador.class` y `EjemploPracticoElevador.java`; `EjemploPracticoElevador.java` se compila en `EjemploPracticoElevador.class`. El estereotipo «ejecutable» especifica que un artefacto es una aplicación, y el estereotipo «archivo» especifica que un artefacto es un archivo que contiene código fuente para el programa ejecutable.

Implementación del controlador

Nuestra simulación implementa los casos de uso "Crear persona" y "Reubicar persona". Hemos estudiado el caso de uso "Reubicar persona" a través del diagrama de actividades de la clase `Persona` en la figura 5.28; implementaremos este caso de uso y el diagrama de actividades en el apéndice E, cuando vayamos a crear la clase `Persona`. Implementaremos el caso de uso "Crear persona" a través de una GUI (interfaz gráfica de usuario). Implementaremos nuestra GUI en la clase `ControladorElevador` (figura 14.23), la cual es una subclase de `JPanel` que contiene dos objetos `JButton`: `primerBotonControlador` (línea 19) y `segundoBotonControlador` (línea 20). Cada objeto `JButton` corresponde a un `Piso` en el cual se va a colocar a una `Persona`.[3] En las líneas 33 a 38 se instancian estos objetos `JButton` y se agregan al `ControladorElevador`.

```
1  // ControladorElevador.java
2  // Controlador para la simulación del elevador
3  package com.deitel.cpej5.elevador.controlador;
4
5  import java.awt.*;
6  import java.awt.event.*;
7
8  import javax.swing.*;
9
10 // paquetes Deitel
11 import com.deitel.cpej5.elevator.modelo.*;
```

Figura 14.23 La clase `ControlElevador` procesa la entrada del usuario. (Parte 1 de 3.)

3. Este método puede realizarse con sólo dos objetos `Piso`. Si el edificio tuviera 100 objetos `Piso`, podríamos optar porque el usuario especificara el `Piso` deseado en un objeto `JTextField`, y que oprimiera un objeto `JButton` para procesar dicha solicitud.


```
12 import com.deitel.cpej5.elevator.evento.*;
13 import com.deitel.cpej5.elevator.ConstantesElevador;
14
15 public class ControladorElevador extends JPanel
16    implements ConstantesElevador {
17
18    // el controlador contiene dos objetos JButton
19    private JButton primerBotonControlador;
20    private JButton segundoBotonControlador;
21
22    // referencia al modelo
23    private SimulacionElevador simulacionElevador;
24
25    public ControladorElevador( SimulacionElevador simulacion )
26    {
27       simulacionElevador = simulacion;
28       setBackground( Color.white );
29
30       // agregar primer botón al controlador
31       primerBotonControlador = new JButton( "Primer piso" );
32       add( primerBotonControlador );
33
34       // agregar segundo botón al controlador
35       segundoBotonControlador = new JButton( "Segundo piso" );
36       add( segundoBotonControlador );
37
38       // clase interna anónima que se registra para eventos ActionEvent de
39       // objeto JButton primerBotonControlador
40       primerBotonControlador.addActionListener(
41          new ActionListener() {
42
43             // se invoca cuando se ha oprimido un objeto JButton
44             public void actionPerformed( ActionEvent evento )
45             {
46                // agregar persona al primer Piso
47                simulacionElevador.agregarPersona(
48                   NOMBRE_PRIMER_PISO );
49
50                // deshabilitar entrada del usuario
51                primerBotonControlador.setEnabled( false );
52             }
53          } // fin de la clase interna anónima
54       );
55
56       // clase interna anónima que se registra para eventos ActionEvent de
57       // objeto JButton segundoBotonControlador
58       segundoBotonControlador.addActionListener(
59          new ActionListener() {
60
61             // se invoca cuando se ha oprimido un objeto JButton
62             public void actionPerformed( ActionEvent evento )
63             {
64                // agregar Persona al segundo Piso
65                simulacionElevador.agregarPersona(
```

Figura 14.23 La clase `ControlElevador` procesa la entrada del usuario. (Parte 2 de 3.)

http://librosysolucionarios.net

```
66                      NOMBRE_SEGUNDO_PISO );
67
68                   // deshabilitar entrada del usuario
69                   segundoBotonControlador.setEnabled( false );
70                }
71             } // fin de la clase interna anónima
72          );
73
74          // clase interna anónima que permite la entrada del usuario en el Piso si
75          // la Persona entra al Elevador en ese Piso
76          simulacionElevador.agregarEscuchaMoverPersona(
77             new EscuchaMoverPersona() {
78
79                // se invoca cuando la Persona ha entrado al Elevador
80                public void personaEntro(
81                   EventoMoverPersona evento )
82                {
83                   // obtener Piso de partida
84                   String ubicacion =
85                      evento.obtenerUbicacion().obtenerNombreUbicacion();
86
87                   // habilitar objeto JButton primerBotonControlador si partió
                        del primer Piso
88                   if ( ubicacion.equals( NOMBRE_PRIMER_PISO ) )
89                      primerBotonControlador.setEnabled( true );
90
91                   // habilitar objeto JButton segundoBotonControlador
                        si partió del segundo Piso
92                   else
93                      segundoBotonControlador.setEnabled( true );
94
95                } // fin del método personaEntro
96
97                // otros métodos que implementan a EscuchaMoverPersona
98                public void personaCreada(
99                   EventoMoverPersona evento ) {}
100
101               public void personaLlego(
102                  EventoMoverPersona evento ) {}
103
104               public void personaSalio(
105                  EventoMoverPersona evento ) {}
106
107               public void personaPartio(
108                  EventoMoverPersona evento ) {}
109
110               public void personaOprimioBoton(
111                  EventoMoverPersona evento ) {}
112
113            } // fin de la clase interna anónima
114         );
115      } // fin del constructor de ControladorElevador
116   }
```

Figura 14.23 La clase **ControlElevador** procesa la entrada del usuario. (Parte 3 de 3.)

En la línea 23 de la clase **ControladorElevador** se declara una referencia al objeto **SimulacionElevador**, ya que el **ControladorElevador** permite al usuario interactuar con el modelo. En las líneas 40 a 54 y 58 a 72 se declaran dos objetos **ActionListener** anónimos y se registran con **primerBotonControlador** y **segundoBotonControlador**, respectivamente, para los eventos **ActionEvent**. Cuando el usuario oprime uno de los dos objetos **JButton**, en las líneas 47 a 48, y 65 a 66 de los métodos **actionPerformed** se hace la llamada al método **agregarPersona** de **SimulacionElevador**, el cual crea una instancia de un objeto **Persona** en el objeto **SimulacionElevador**, en el **Piso** especificado. El método **agregarPersona** toma como argumento una variable **String** declarada en la interfaz **ConstantesElevador** (Fig. 14.24). Esta interfaz (utilizada por clases como **ControladorElevador**, **SimulacionElevador**, **Elevador**, **Piso** y **VistaElevador**) proporciona las constantes que especifican los nombres de los objetos **Ubicacion** en nuestra simulación.

```
1  // ConstantesElevador.java
2  // Constantes utilizadas entre ModeloElevador y VistaElevador
3  package com.deitel.cpej5.elevador;
4
5  public interface ConstantesElevador {
6
7     public static final String NOMBRE_PRIMER_PISO = "primerPiso";
8     public static final String NOMBRE_SEGUNDO_PISO = "segundoPiso";
9     public static final String NOMBRE_ELEVADOR = "elevador";
10 }
```

Figura 14.24 La interfaz **ConstantesElevador** proporciona las constantes para los nombres de los objetos **Ubicacion**.

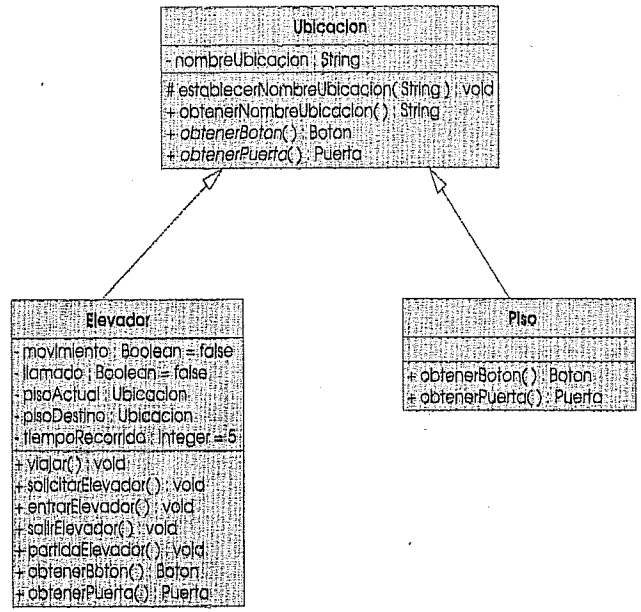


Figura 14.25 Diagrama de clases modificado que muestra la generalización de la superclase **Ubicacion**, y las subclases **Elevador** y **Piso**.

http://librosysolucionarios.net

En las líneas 51 y 69 de los métodos **actionPerformed** se deshabilitan los objetos **JButton** respectivos, para evitar que el usuario cree más de una **Persona** por cada **Piso**. En las líneas 76 a 114 de la clase **ControladorElevador** se declara un nuevo objeto anónimo **EscuchaMoverPersona**, el cual se registra con el objeto **SimulacionElevador** para rehabilitar los objetos **JButton**. El método **personaEntro** (líneas 80 a 95) de la clase **EscuchaMoverPersona** rehabilita el objeto **JButton** asociado con el **Piso** al que el **Elevador** da servicio; una vez que la **Persona** ha entrado al **Elevador**, el usuario puede colocar a otra **Persona** en el **Piso**.

En la sección 10.11 mencionamos que las clases **Elevador** y **Piso** heredaban el atributo **capacidad** de la superclase **Ubicacion**; en el apéndice H íbamos a utilizar este atributo para evitar que más de una **Persona** ocupara la misma **Ubicacion**. Sin embargo, el método **personaEntro** de **EscuchaMoverPersona** en la clase **ControladorElevador** evita que el usuario cree más de una **Persona** por **Piso**. Por lo tanto, ya no necesitamos el atributo **capacidad** en la clase **Ubicacion**. En la figura 14.25 se muestra el diagrama de clases modificado de la figura 10.25, sin este atributo.

Implementación: EjemploPracticoElevador.java

Utilizaremos los diagramas de las figuras 14.22, 14.21 y 13.28 para implementar la clase **EjemploPracticoElevador** (figura 14.26). En las líneas 12 a 14 se importan los paquetes **modelo**, **vista** y **controlador** según lo especificado en la figura 14.22.

```
1  // EjemploPracticoElevador.java
2  // Aplicación con modelo, vista y controlador (MVC) del elevador
3  package com.deitel.cpej5.elevador;
4
5  // paquetes básicos de Java
6  import java.awt.*;
7
8  // paquetes de extensión de Java
9  import javax.swing.*;
10
11 // paquetes Deitel
12 import com.deitel.cpej5.elevador.modelo.*;
13 import com.deitel.cpej5.elevador.vista.*;
14 import com.deitel.cpej5.elevador.controlador.*;
15
16 public class EjemploPracticoElevador extends JFrame {
17
18    // modelo, vista y controlador
19    private SimulacionElevador modelo;
20    private VistaElevador vista;
21    private ControladorElevador controlador;
22
23    // el constructor instancia a modelo, vista y controlador
24    public EjemploPracticoElevador()
25    {
26       super( "Simulación de elevador de Deitel" );
27
28       // instanciar a modelo, vista y controlador
29       modelo = new SimulacionElevador();
30       vista = new VistaElevador();
31       controlador = new ControladorElevador( modelo );
32
33       // registrar a Vista para eventos de Modelo
```

Figura 14.26 La clase **EjemploPracticoElevador** es la aplicación para la simulación del elevador. (Parte 1 de 2.)

http://librosysolucionarios.net


```
34       modelo.establecerEscuchaSimulacionElevador( vista );
35
36       // agregar vista y controlador a SimulacionElevador
37       getContentPane().add( vista, BorderLayout.CENTER );
38       getContentPane().add( controlador, BorderLayout.SOUTH );
39
40    } // fin del constructor de SimulacionElevador
41
42    // el método main inicia el programa
43    public static void main( String args[] )
44    {
45       // instanciar a SimulacionElevador
46       EjemploPracticoElevador ejemploPractico = new EjemploPracticoElevador();
47       ejemploPractico.setDefaultCloseOperation( EXIT_ON_CLOSE );
48       ejemploPractico.pack();
49       ejemploPractico.setVisible( true );
50    }
51 }
```

Figura 14.26 La clase **EjemploPracticoElevador** es la aplicación para la simulación del elevador. (Parte 2 de 2.)

En la figura 14.21 se especifica que **EjemploPracticoElevador** es una subclase de **JFrame**; en la línea 16 se declara a la clase **EjemploPracticoElevador** como una clase **public** que extiende a la clase **JFrame**. En las líneas 19 a 21 se implementa la agregación de la clase **SimulacionElevador** de **EjemploPracticoElevador**, la **VistaElevador** y el **ControladorElevador** (mostrados en la figura 14.21) al declarar un objeto de cada clase. En las líneas 29 a 31 del constructor de **EjemploPracticoElevador** se inicializan estos objetos.

En las figuras 14.21 y 14.22 se especifica que **VistaElevador** es un componente de escucha **EscuchaSimulacionElevador** para la **SimulacionElevador**. En la línea 34 se registra el objeto **VistaElevador** como un componente de escucha para eventos **EventoSimulacionElevador**, por lo que el objeto **VistaElevador** puede recibir eventos del objeto **SimulacionElevador** y representar apropiadamente el estado del modelo. En las líneas 37 a 38 se agregan los objetos **VistaElevador** y **ControladorElevador** al objeto **EjemploPracticoElevador**. De acuerdo con los estereotipos de la figura 14.22, **EjemploPracticoElevador.java** se compila en **EjemploPracticoElevador.class**, que es ejecutable; en las líneas 43 a 50 se proporciona el método **main** que ejecuta la aplicación.

En la sección 16.11 concluiremos el diseño del modelo, al resolver los problemas de interacción que encontramos en la figura 10.26. Finalmente, en la sección 19.7 completaremos el diseño de la vista, y describiremos con mayor detalle cómo el objeto **VistaElevador** recibe eventos del objeto **SimulacionElevador**. Estas últimas dos secciones lo prepararán para el recorrido de la implementación de nuestra simulación de elevador en los apéndices D a F.

14.14 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.awt` y `javax.swing`

Ahora continuaremos con nuestra discusión de la sección 1.16, acerca de los patrones de diseño. En esta sección le presentaremos los patrones de diseño asociados con los componentes de la GUI de Java. Después de leer esta sección, deberá entender mejor cómo estos componentes sacan ventaja de los patrones de diseño, y cómo los desarrolladores integran los patrones de diseño con aplicaciones de la GUI de Java.

14.14.1 Patrones de diseño de creación

Ahora continuaremos con nuestro análisis de los patrones de diseño de creación, los cuales proporcionan formas de instanciar objetos en un sistema.

Método de fábrica

Suponga que diseñaremos un sistema para abrir una imagen desde un archivo especificado. Existen varios formatos de imagen distintos, como GIF y JPEG. Podemos utilizar el método `createImage` de la clase `java.awt.Component` para crear un objeto `Image`. Por ejemplo, para crear una imagen JPEG y GIF en un objeto de una subclase de `Component` (como un objeto `JPanel`) debemos pasar el nombre del archivo de imagen al método `createImage`, el cual devuelve un objeto `Image` que almacena los datos de la imagen. Podemos crear dos objetos `Image`, cada uno de los cuales contiene datos para dos imágenes que tienen estructuras completamente distintas. Por ejemplo, una imagen JPEG puede almacenar hasta 16.7 millones de colores, mientras que una imagen GIF puede almacenar sólo hasta 256. Además, una imagen GIF puede contener píxeles transparentes que no se despliegan en pantalla, mientras que una imagen JPEG no puede contener píxeles transparentes.

La clase `Image` es una clase abstracta que representa a una imagen que podemos mostrar en la pantalla. Utilizando el parámetro que le pasa el programador, el método `createImage` determina la subclase de `Image` específica, a partir de la cual se va a instanciar el objeto `Image`. Podemos diseñar sistemas para permitir al usuario que especifique cuál imagen crear, y el método `createImage` determinará la subclase a partir de la cual se va a instanciar el objeto `Image`. Si el parámetro que se pasa al método `createImage` hace referencia a un archivo JPEG, el método `createImage` instancia y devuelve un objeto de una subclase de `Image` apropiada para imágenes JPEG. Si el parámetro hace referencia a un archivo GIF, `createImage` instancia y devuelve un objeto de una subclase de `Image` apropiada para imágenes GIF.

El método `createImage` es un ejemplo del *patrón de diseño Método de fábrica*. El único propósito de este *método de fábrica* es crear objetos, permitiendo al sistema determinar cuál clase instanciar en tiempo de ejecución. Podemos diseñar un sistema que permita a un usuario especificar el tipo de imagen a crear en tiempo de ejecución. La clase `Componente` tal vez no pueda determinar qué subclase de `Image` instanciar, sino hasta que el usuario especifique la imagen a cargar. Para obtener más información acerca del método `createImage`, visite la página:

```
www.java.sun.com/j2se/1.4/docs/api/java/awt/Component.html#
   createImage(java.awt.image.ImageProducer)
```

14.14.2 Patrones de diseño de estructura

Ahora veremos otros tres patrones de diseño de estructura. El patrón de diseño Adaptador ayuda a los objetos con interfaces incompatibles a colaborar entre sí. El patrón de diseño Puente ayuda a los diseñadores a mejorar la independencia de plataformas en sus sistemas. El patrón de diseño Compuesto proporciona una manera para que los diseñadores organicen y manipulen objetos.

Adaptador

El *patrón de diseño Adaptador* proporciona un objeto con una nueva interfaz que se *adapta* a la interfaz de otro objeto, permitiendo que ambos objetos colaboren entre sí. El adaptador en este patrón es similar a un adaptador para un enchufe en un aparato eléctrico; los contactos eléctricos en Europa tienen una forma distinta de los de Estados Unidos, por lo cual se necesita un adaptador para conectar un aparato estadounidense en un contacto eléctrico europeo, y viceversa.

Java proporciona varias clases que utilizan el patrón de diseño Adaptador. Los objetos de las subclases concretas de estas clases actúan como adaptadores entre los objetos que generan ciertos eventos, y los objetos que manejan esos eventos. Por ejemplo, un objeto `MouseAdapter` (que explicamos en la sección 13.13) adapta a un objeto que genera eventos `MouseEvent` a un objeto que maneja eventos `MouseEvent`.

Puente

Suponga que diseñaremos la clase `Boton`, tanto para el sistema operativo Windows como para el sistema operativo Macintosh. La clase `Boton` contiene información específica sobre botones, tal como un objeto `ActionListener` y una etiqueta. Diseñamos las clases `BotonWin32` y `BotonMac` para que extiendan a la clase `Boton`. La clase `BotonWin32` contiene la información de "apariencia visual" acerca de cómo mostrar un objeto `Boton` en el sistema operativo Windows, y la clase `BotonMac` contiene la información de "apariencia visual" acerca de cómo mostrar un objeto `Boton` en el sistema operativo Macintosh.

http://librosysolucionarios.net

De este método surgen dos problemas. En primer lugar, si creamos nuevas subclases de `Boton`, debemos crear las subclases correspondientes para `BotonWin32` y `BotonMac`. Por ejemplo, si creamos la clase `BotonImagen` (un boton con un objeto `Imagen` superpuesto) que extienda a la clase `Boton`, debemos crear las subclases adicionales `BotonImagenWin32` y `BotonImagenMac`. De hecho, debemos crear subclases de `Boton` para cada uno de los sistemas operativos que se vayan a soportar, lo cual incrementa el tiempo de desarrollo. El segundo problema es que, cuando un nuevo sistema operativo entre al mercado, tendremos que crear subclases de `Boton` adicionales que sean específicas para ese sistema operativo.

El *patrón de diseño Puente* evita estos problemas al dividir una abstracción (es decir, una clase `Boton`) y sus implementaciones (como, `BotonWin32`, `BotonMac`, etc.) en jerarquías de clases separadas. Por ejemplo, las clases de AWT de Java utilizan el patrón de diseño Puente para permitir a los diseñadores crear subclases de `Boton` de AWT sin necesidad de crear subclases específicas para cada sistema operativo. Cada `Boton` de AWT mantiene una referencia a un objeto `BotonIgual`, que es la superclase para implementaciones en plataformas específicas, como `BotonWin32Igual`, `BotonMacIgual`, etcétera. Cuando un programador crea un objeto `Boton`, la clase `Boton` llama al método de fábrica `crearBoton` de la clase `KitHerramientas` para crear el objeto `BotonIgual` de la plataforma específica. El objeto `Boton` guarda una referencia a su objeto `BotonIgual`; esta referencia es el "puente" en el patrón de diseño Puente. Cuando el programador invoca a los métodos en el objeto `Boton`, éste delega el trabajo al método apropiado de nivel más bajo en su clase `BotonIgual` para satisfacer la solicitud. Si un diseñador crea una subclase de `Boton` llamada `BotonImagen`, no necesita crear su correspondiente clase `BotonImagenWin32` o `BotonImagenMac` con capacidades de dibujo de imágenes específicas de cada plataforma. Un objeto `BotonImagen` "es un" objeto `Boton`. Por lo tanto, cuando un objeto `BotonImagen` necesita mostrar su imagen, utiliza el objeto `Graphics` de su objeto `BotonIgual` para desplegar la imagen en cada una de las plataformas. Este patrón de diseño permite a los diseñadores crear nuevos componentes de la GUI para todas las plataformas, utilizando un "puente" para ocultar los detalles específicos de cada plataforma.



Tip de portabilidad 14.3

Los diseñadores a menudo utilizan el patrón de diseño Puente para mejorar la independencia de plataformas en sus sistemas. Este patrón de diseño permite a los diseñadores crear nuevos componentes para todas las plataformas, utilizando un "puente" para ocultar los detalles específicos de cada plataforma.

Compuesto

A menudo, los diseñadores organizan los componentes en estructuras jerárquicas (es decir, una jerarquía de directorios y archivos en un sistema de archivos); cada nodo en la estructura representa a un componente (por ejemplo, un archivo o directorio). Cada nodo puede contener referencias a otros nodos. A un nodo se le llama *rama* si contiene una referencia a uno o más nodos (por ejemplo, un directorio que contiene archivos). A un nodo se le llama *hoja* si contiene una referencia a otro nodo (como, un archivo). En ocasiones, una estructura contiene objetos de varias clases distintas (por ejemplo, un directorio puede contener archivos y directorios). Cuando un objeto (llamado *cliente*) desea recorrer la estructura, el cliente debe determinar la clase específica para cada nodo. Hacer esta determinación lleva tiempo, y la estructura puede ser difícil de mantener.

En el *patrón de diseño Compuesto*, cada componente en una estructura jerárquica implementa a la misma interfaz o extiende a una superclase común. Este polimorfismo (presentado en el capítulo 1) asegura que los clientes puedan recorrer todos los elementos (rama u hoja) de manera uniforme en la estructura. Mediante el uso de este patrón, un cliente que recorra la estructura no tendrá qué determinar el tipo de cada componente, ya que todos los componentes implementan a la misma interfaz o extienden a la misma superclase.

Los componentes de la GUI de Java utilizan el patrón de diseño Compuesto. Considere como ejemplo la clase `JPanel`, que es un componente de Swing y extiende a la clase `JComponent`. La clase `JComponent` extiende a la clase `java.awt.Container`, la cual a su vez extiende a la clase `java.awt.Component` (figura 14.27). La clase `Container` proporciona el método `add`, el cual anexa un objeto `Component` (o al objeto de una subclase de `Component`) a ese objeto `Container`. Por lo tanto, puede agregarse un objeto `JPanel` a cualquier objeto de una subclase `Component`, y cualquier objeto de una subclase `Component` puede agregarse a ese objeto `JPanel`. Un objeto `JPanel` puede contener cualquier componente de la GUI, sin necesidad de saber el tipo específico de cada componente. Casi todas las clases de GUI son tanto contenedores como componentes, con lo cual permiten una anidación y estructuración arbitrariamente compleja de las GUIs.

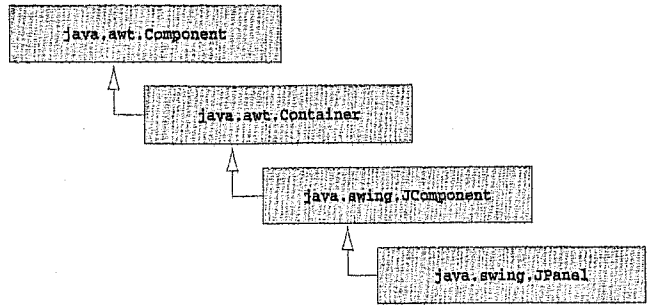


Figura 14.27 Jerarquía de herencia para la clase `JPanel`.

Un cliente tal como un objeto `JPanel` puede recorrer, uniformemente, todos los componentes en la jerarquía. Por ejemplo, si el objeto `JPanel` llama al método `repaint` de la superclase `Container`, el método `repaint` mostrará a ese objeto `JPanel` junto con todos los componentes que se hayan agregado al objeto `JPanel`. El método `repaint` no tiene que determinar el tipo de cada componente, ya que todos los componentes heredan de la superclase `Container`, la cual contiene el método `repaint`.

14.14.3 Patrones de diseño de comportamiento

En esta sección continuaremos con nuestra discusión sobre los patrones de diseño de comportamiento. Hablaremos sobre los patrones de diseño Cadena de responsabilidad, Comando, Observador, Estrategia y Método de plantilla.

Cadena de responsabilidad

En los sistemas orientados a objetos, los objetos interactúan enviando mensajes unos a otros. A menudo, un sistema necesita determinar, en tiempo de ejecución, el objeto que manejará un mensaje específico. Por ejemplo, considere el diseño de un sistema telefónico de oficina con tres líneas. Cuando una persona llama a la oficina, la primera línea se encarga de la llamada; si la primera línea está ocupada, la segunda línea se encarga de la llamada y, si esta línea está ocupada, la tercera línea es la que se encarga de la llamada. Si todas las líneas en el sistema están ocupadas, una voz automatizada indica a la persona que espere a que una de las líneas esté disponible; cuando se libera una línea, ésta se encarga de la llamada.

El patrón de diseño *Cadena de responsabilidad* permite a un sistema determinar, en tiempo de ejecución, el objeto que se encargará de un mensaje. Este patrón permite a un objeto enviar un mensaje a varios objetos en una *cadena* de objetos. Cada objeto en la cadena puede manejar el mensaje o pasarlo al siguiente objeto en la cadena. Por ejemplo, la primera línea en el sistema telefónico es el primer objeto en la cadena de responsabilidad, la segunda línea es el segundo objeto, la tercera es el tercer objeto y la voz automatizada es el cuarto objeto. Observe que este mecanismo no es el objeto final en la cadena; la siguiente línea disponible se encarga del mensaje, y esa línea es el objeto final en la cadena. La cadena se crea dinámicamente, en respuesta a la presencia o ausencia de manejadores de mensajes específicos.

Varios componentes de la GUI de AWT de Java utilizan el patrón de diseño Cadena responsabilidad para manejar ciertos eventos. Por ejemplo, la clase `java.awt.Button` sobrescribe al método `processEvent` de la clase `java.awt.Component` para procesar objetos `AWTEvent`. El método `processEvent` trata de hacerse cargo del evento `AWTEvent` al momento de recibirlo como un argumento. Si el método `processEvent` determina que el evento `AWTEvent` es un `ActionEvent` (es decir, que se haya oprimido el objeto `Boton`), el método se hace cargo del evento invocando al método `processActionEvent`, el cual informa a cualquier objeto `ActionListener` registrado con el objeto `Boton`, que ese objeto `Boton` se ha oprimido. Si el método `processEvent` determina que el evento `AWTEvent` no es un `ActionEvent`, el método no podrá manejar ese evento y lo pasará al método `processEvent` de la superclase `Component`, la cual pasa el evento al siguiente componente de escucha en la cadena).

http://librosysolucionarios.net

Comando

Las aplicaciones a menudo proporcionan a los usuarios varias maneras de llevar a cabo una tarea dada. Por ejemplo, en un procesador de palabras podría haber un menú **Edición** con elementos de menú para cortar, copiar y pegar texto. También podría haber una barra de herramientas o un menú contextual ofreciendo los mismos elementos. La funcionalidad que proporciona la aplicación es la misma en cada caso; los distintos componentes de la interfaz para invocar la funcionalidad se proporcionan para la conveniencia del usuario. Sin embargo, la misma instancia del componente de la GUI (por ejemplo, `JButton`) no puede usarse para menús, barras de herramientas y menús contextuales, por lo que el desarrollador debe codificar la misma funcionalidad tres veces. Si hubiera muchos de esos elementos de la interfaz, repetir esta funcionalidad sería un proceso tedioso y propenso a errores.

El *patrón de diseño Comando* resuelve este problema al permitir a los desarrolladores encapsular la funcionalidad deseada (por ejemplo, copiar texto) una vez en un objeto reutilizable; esa funcionalidad puede entonces agregarse a un menú, barra de herramientas, menú contextual o cualquier otro mecanismo. Este patrón de diseño se conoce como Comando, ya que define un comando (o instrucción) a ejecutar. Este patrón permite a un diseñador encapsular un comando, de manera que éste pueda utilizarse entre varios objetos.

Observador

Suponga que deseamos diseñar un programa para ver la información de cuentas bancarias. Este sistema incluye la clase `DatosEstadoCuentaBancario` para almacenar datos pertenecientes a los estados de cuenta bancarios, y las clases `TextoPantalla`, `GraficoBarrasPantalla` y `GraficoPastelPantalla` para mostrar los datos.[4] En la figura 14.28 se muestra el diseño de nuestro sistema. La clase `TextoPantalla` muestra los datos en formato de texto, la clase `GraficoBarrasPantalla` muestra los datos en formato de gráfico de barras y la clase `GraficoPastelPantalla` muestra los datos en forma de gráfico de pastel. Queremos diseñar el sistema de manera que el objeto `DatosEstadoCuentaBancario` notifique a los objetos que muestran los datos, cuando haya un cambio en estos datos. También queremos diseñar el sistema para reducir el *acoplamiento* (el grado al que las clases dependen, unas de otras, en un sistema).



Observación de ingeniería de software 14.2

Las clases con poco acoplamiento son más fáciles de reutilizar y modificar que las clases con un acoplamiento estrecho, las cuales dependen considerablemente unas de otras. Una modificación en una clase de un sistema con mucho acoplamiento ocasiona que se tengan que modificar otras clases en ese sistema. Una modificación a una clase de un grupo de clases con poco acoplamiento requeriría de hacer pocas o ninguna modificación a las demás clases en el grupo.

El *patrón de diseño Observador* es apropiado para sistemas como el de la figura 14.28. Este patrón promueve el poco acoplamiento entre un objeto *sujeto* y objetos *observadores*; un sujeto notifica a los observadores cuando éste cambia de estado. Al ser notificados por el sujeto, los observadores cambian en respuesta al cambio en el sujeto. En nuestro ejemplo, el objeto `DatosEstadoCuentaBancario` es el sujeto y los objetos que muestran los datos son los observadores. Un sujeto puede notificar a varios observadores; por lo tanto, el sujeto tiene una relación de uno a varios con los observadores.

La API de Java contiene clases que utilizan el patrón de diseño Observador. La clase `java.util.Observable` representa a un sujeto. La clase `Observable` proporciona el método `addObserver`, el cual toma un argumento `java.util.Observer`. La interfaz `Observer` permite al objeto `Observable` notificar al objeto `Observer` cuando el objeto `Observable` cambia de estado. El objeto `Observer` puede ser una instancia de cualquier clase que implemente a la interfaz `Observer`; ya que el objeto `Observable` invoca a los métodos declarados en la interfaz `Observer`, los objetos permanecen con poco acoplamiento. Si un desarrollador cambia la forma en que un objeto `Observer` específico responde a los cambios en el objeto `Observable`, el desarrollador no necesita cambiar el objeto `Observable`. El objeto `Observable` interactúa con sus objetos `Observer` sólo a través de la interfaz `Observer`, que permite el poco acoplamiento.

El ejemplo práctico opcional de la simulación del elevador en las secciones "Acerca de los objetos" utiliza el patrón de diseño Observador para permitir al objeto `ModeloElevador` (el sujeto) notificar al objeto `Vis-`

4. Este ejemplo es la base para el patrón de la arquitectura Modelo-vista-controlador, el cual describimos en las secciones 13.17 y 17.11.

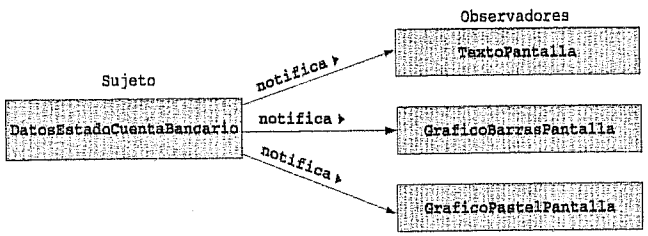


Figura 14.28 Base para el patrón de diseño Observador.

`taElevador` (el observador) cuando haya cambios en el objeto `ModeloElevador`. La simulación no utiliza la clase `Observable` ni la interfaz `Observer` de la biblioteca de Java; en vez de ello, utiliza una interfaz personalizada llamada `EscuchaModeloElevador` que proporciona una funcionalidad similar a la de la interfaz `Observable`.

Los componentes de la GUI de Swing utilizan el patrón de diseño Observador. Los componentes de la GUI colaboran con sus componentes de escucha para responder a las interacciones del usuario. Por ejemplo, para observar los cambios de estado en un objeto `JButton` (el sujeto), un objeto `ActionListener` se registra para manejar los eventos de ese objeto `JButton`. Cuando lo oprime el usuario, el objeto `JButton` notifica a sus objetos `ActionListener` (los observadores) que el estado del objeto `JButton` ha cambiado (es decir, el objeto `JButton` ha sido oprimido).

Estrategia

El *patrón de diseño Estrategia* es similar al patrón de diseño Estado (el cual describimos en la sección 10.12). Anteriormente mencionamos que el patrón de diseño Estado contiene un objeto de estado, el cual encapsula el estado de un objeto de contexto. El patrón de diseño Estrategia contiene un *objeto de estrategia*, el cual es análogo al objeto de estado del patrón de diseño Estado. La diferencia clave entre un objeto de estado y un objeto de estrategia es que el objeto de estado encapsula a un *algoritmo*, y no a la información de estado.

Por ejemplo, los componentes de `java.awt.Container` implementan el patrón de diseño Estrategia utilizando objetos `LayoutManager` (descritos en la sección 13.15) como objetos de estrategia. En el paquete `java.awt`, las clases `FlowLayout`, `BorderLayout` y `GridLayout` implementan a la interfaz `LayoutManager`. Cada una de las clases utiliza el método `addLayoutComponent` para agregar componentes de la GUI a un objeto `Container`. Sin embargo, cada método utiliza un algoritmo distinto para mostrar estos componentes de la GUI. Un esquema `FlowLayout` muestra a los componentes en una secuencia de izquierda a derecha, un esquema `BorderLayout` muestra a los componentes en cinco regiones, y un esquema `GridLayout` muestra a los componentes en formato de filas y columnas.

La clase `Container` contiene una referencia a un objeto `LayoutManager` (el objeto de estrategia). La referencia a una interfaz (es decir, la referencia al objeto `LayoutManager`) puede guardar referencias a objetos de clases que implementen a esa interfaz (es decir, los objetos `FlowLayout`, `BorderLayout` o `GridLayout`), por lo que el objeto `LayoutManager` puede hacer referencia a un objeto `FlowLayout`, `BorderLayout` o `GridLayout` en cualquier momento dado. La clase `Container` puede cambiar esta referencia mediante el método `setLayout`, para seleccionar distintos esquemas en tiempo de ejecución.

La clase `DemoFlowLayout` (figura 13.24) demuestra la aplicación del patrón Estrategia; en la línea 16 se declara un nuevo objeto `FlowLayout` y en la línea 19 se invoca al método `setLayout` del objeto `Container` para asignar el objeto `FlowLayout` al objeto `Container`. En este ejemplo, el objeto `FlowLayout` proporciona la estrategia para distribuir los componentes en la pantalla.

Método de plantilla

El *patrón de diseño Método de plantilla* también trata con los algoritmos. El patrón de diseño Estrategia permite que varios objetos contengan distintos algoritmos. No obstante, el patrón de diseño Método de plantilla quiere que todos los objetos compartan un solo algoritmo, definido por una superclase.

Por ejemplo, considere el diseño de la figura 14.28, el cual mencionamos en la discusión sobre el patrón de diseño Observador. Los objetos de las clases `TextoPantalla`, `GraficoBarrasPantalla` y `GraficoPastelPantalla` utilizan el mismo algoritmo básico para adquirir y mostrar los datos: obtener todas las instrucciones del objeto `DatosEstadoCuentaBancario`, analizar las instrucciones y luego mostrarlas. El patrón de diseño Método plantilla nos permite crear una superclase abstracta llamada `EstadoCuentaBancarioPantalla`, la cual proporciona el algoritmo común para mostrar los datos. En este ejemplo, el algoritmo invoca a los métodos abstractos `obtenerDatos`, `analizarDatos` y `mostrarDatos`. Las clases `TextoPantalla`, `GraficoBarrasPantalla` y `GraficoPastelPantalla` extienden a la clase `EstadoCuentaBancarioPantalla` para heredar el algoritmo, de manera que cada objeto puede utilizar el mismo algoritmo. Cada subclase de `EstadoCuentaBancarioPantalla` sobrescribe entonces a cada método, de una manera específica a esa subclase, ya que cada clase implementa al algoritmo de manera distinta. Por ejemplo, las clases `TextoPantalla`, `GraficoBarrasPantalla` y `GraficoPastelPantalla` podrían obtener y analizar los datos de manera similar, pero cada clase muestra esos datos de manera distinta.

El patrón de diseño Método de plantilla nos permite extender el algoritmo hacia otras subclases de `EstadoCuentaBancarioPantalla`; por ejemplo, podríamos crear clases como `GraficoLinealPantalla` o la clase `Pantalla3Dimensional`, que utilicen el mismo algoritmo heredado de la clase `EstadoCuentaBancarioPantalla` y proporcionen distintas implementaciones de los métodos abstractos a los que llame el algoritmo.

14.14.4 Conclusión

En esta sección "Descubrimiento de los patrones de diseño" hablamos sobre cómo los componentes de Swing aprovechan los patrones de diseño, y cómo los desarrolladores pueden integrar los patrones de diseño con aplicaciones de GUI en Java. En la sección "Descubrimiento de los patrones de diseño" 16.12 hablaremos sobre los patrones de diseño de concurrencia, que son especialmente útiles para desarrollar sistemas con subprocesamiento múltiple.

RESUMEN

- Un objeto `JTextArea` proporciona un área para manipular varias líneas de texto. Al igual que la clase `JTextField`, la clase `JTextArea` extiende a `JTextComponent`. Generalmente se utiliza un evento externo para indicar cuándo debe procesarse el texto en un objeto `JTextArea`. Para proporcionar barras de desplazamiento a un objeto `JTextArea`, se le adjunta un objeto `JScrollPane`. El método `getSelectedText` devuelve el texto seleccionado de un objeto `JTextArea`. El método `setText` establece el texto en un objeto `JTextArea`.
- Las políticas de barra de desplazamiento horizontal y vertical para un objeto `JScrollPane` se establecen cuando se crea este objeto, o con los métodos `setHorizontalScrollBarPolicy` y `setVerticalScrollBarPolicy` de la clase `JScrollPane`.
- Para proporcionar la envoltura automática de líneas en un objeto `JTextArea`, se debe adjuntar a un objeto `JScrollPane` con la política de barra de desplazamiento horizontal `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`.
- Un objeto `JPanel` puede utilizarse como área dedicada de dibujo, la cual puede recibir eventos de ratón y a menudo se extiende para crear nuevos componentes de la GUI.
- Los componentes de Swing que heredan de la clase `JComponent` contienen el método `paintComponent`, el cual les ayuda a dibujar apropiadamente en el contexto de una GUI ligera de Swing. El método `paintComponent` debe sobrescribirse para llamar al método `paintComponent` de la superclase como la primera instrucción en su cuerpo.
- Las clases `JFrame` y `JApplet` no son subclases de `JComponent`; por lo tanto, no contienen el método `paintComponent` (tienen el método `paint`).
- Al llamar a `repaint` para un componente de Swing, se indica que ese componente deberá pintarse lo más pronto posible. El fondo del componente de la GUI se borra solamente si este componente es opaco. La mayoría de los componentes de Swing son transparentes de manera predeterminada. El método `setOpaque` de `JComponent` puede recibir un argumento `boolean` que indique si el componente es opaco (`true`) o transparente (`false`).
- El método `setTitle` muestra una cadena en la barra de título de una ventana.
- Todo dibujo en cualquier componente de la GUI se realiza con coordenadas que se miden desde la esquina superior izquierda (0, 0) de ese componente de la GUI.

http://librosysolucionarios.net

- A menudo, los administradores de esquemas utilizan el método `getPreferredSize` de un componente de la GUI para determinar la anchura y altura preferidas del componente, cuando éste se distribuya en pantalla como parte de una GUI. Si un nuevo componente tiene una anchura y altura preferidas, debe sobrescribir el método `getPreferredSize` para devolver esa anchura y altura como un objeto de la clase `Dimension` (paquete `java.awt`).
- El tamaño predeterminado de un objeto `JPanel` es 10 píxeles de ancho y 10 píxeles de largo.
- Una operación de arrastre del ratón empieza con un evento de oprimir un botón del ratón. Todos los eventos de arrastre de ratón subsiguientes se envían al componente de la GUI que recibió el evento original de oprimir un botón del ratón.
- Los objetos `JSlider` permiten al usuario seleccionar de entre un rango de valores enteros. Los objetos `Jslider` pueden mostrar marcas más distanciadas, marcas menos distanciadas y etiquetas para las marcas. También soportan el *ajuste a la marca*, en el que al colocar el indicador entre dos marcas, éste se *ajusta* a la marca más cercana.
- Si un objeto `JSlider` tiene el enfoque, la tecla de flecha izquierda y la tecla de flecha derecha hacen que el indicador del objeto `JSlider` se decremente o incremente en 1. La tecla de flecha hacia abajo y la tecla de flecha hacia arriba también hacen que el indicador del objeto `JSlider` se decremente o incremente en 1 marca, respectivamente. Las teclas *Av Pág* (avance de página) y *Re Pág* (retroceso de página) hacen que el indicador del objeto `JSlider` se decremente o incremente en incrementos de bloque de una décima parte del rango de valores, respectivamente. La *tecla Inicio* desplaza el indicador hacia el valor mínimo del objeto `JSlider`, y la *tecla Fin* desplaza el indicador hacia el valor máximo del objeto `JSlider`.
- Los objetos `JSlider` tienen una orientación horizontal o vertical. Para un objeto `JSlider` horizontal, el valor mínimo se encuentra en el extremo izquierdo y el valor máximo se encuentra en el extremo derecho. Para un objeto `JSlider` vertical, el valor mínimo se encuentra en el extremo inferior y el valor máximo se encuentra en el extremo superior. La posición del indicador muestra el valor actual del objeto `JSlider`. El método `getValue` de la clase `JSlider` devuelve la posición actual del indicador.
- El método `setMajorTickSpacing` de la clase `JSlider` establece el espaciado para las marcas en un objeto `JSlider`. El método `setPaintTicks` con un argumento `true` indica que deben mostrarse las marcas.
- Los objetos `JSlider` generan eventos `ChangeEvent` cuando el usuario interactúa con ellos. Un objeto `ChangeListener` declara el método `stateChanged`, el cual puede responder a eventos `ChangeEvent`.
- Un objeto `JFrame` es una ventana con una barra de título y un borde. El método `setDefaultCloseOperation` de `JFrame` especifica la acción a realizar cuando el usuario cierra la ventana.
- Una ventana no se muestra en pantalla sino hasta que su método `setVisible` sea llamado, con `true` como argumento.
- El tamaño de una ventana debe establecerse mediante una llamada a `setSize`. La posición que tendrá la ventana cuando aparezca en la pantalla se especifica mediante el método `setLocation`.
- Todas las ventanas generan eventos de ventana cuando el usuario las manipula. La interfaz `WindowListener` proporciona siete métodos manejadores de eventos: `windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowIconified`, `windowDeiconified` y `windowOpened`.
- Los menús son una parte integral de las GUIs; ya que permiten al usuario realizar acciones sin "atestar" innecesariamente una interfaz gráfica de usuario con componentes de la GUI adicionales. En las GUIs de Swing, los menús pueden adjuntarse solamente a los objetos de clases que proporcionen el método `setJMenuBar` (tales como `JFrame` y `JApplet`).
- Las clases utilizadas para declarar menús son `JMenuBar`, `JmenuItem`, `JMenu`, `JCheckBoxMenuItem` y la clase `JRadioButtonMenuItem`. Un objeto `JMenuBar` es un contenedor de menús. Un objeto `JMenuItem` es un componente de la GUI que está dentro de un menú y que, cuando se selecciona, ocasiona que se realice una acción. Un objeto `JMenu` contiene elementos de menú y pueden agregarse a objetos `JMenuBar` o a otros objetos `JMenu`, en forma de submenús. Cuando se hace clic en un menú, éste se expande para mostrar su lista de elementos de menú. El método `addSeparator` de `JMenu` agrega una línea separadora a un menú.
- Cuando se selecciona un objeto `JCheckBoxMenuItem`, aparece una marca de verificación a la izquierda del elemento de menú. Cuando se selecciona de nuevo el mismo objeto `JCheckBoxMenuItem`, se quita la marca de verificación a la izquierda del elemento de menú.
- Cuando se mantienen varios objetos `JRadioButtonMenuItem` como parte de un objeto `ButtonGroup`, sólo puede seleccionarse un elemento en el grupo a la vez. Cuando se selecciona un objeto `JRadioButtonMenuItem`, aparece un círculo relleno a la izquierda del elemento de menú. Cuando se selecciona otro objeto `JCheckBoxMenuItem`, se quita el círculo relleno a la izquierda del elemento de menú previamente seleccionado.
- El método `setMnemonic` de `AbstractButton` especifica el nemónico para un objeto `AbstractButton`. Los caracteres nemónicos generalmente se muestran subrayados.
- Los cuadros de diálogo generalmente son modales. Un cuadro de diálogo modal no permite el acceso a ninguna de las otras ventanas en la aplicación, sino hasta que se cierre. Los cuadros de diálogo que se muestran con la clase `JOptionPane` son cuadros de diálogo modales. La clase `JDialog` puede usarse para crear sus propios cuadros de diálogo modales o no modales.

- Los menús contextuales sensibles al contexto se crean con la clase `JPopupMenu`. En la mayoría de los sistemas, el evento de desencadenamiento de menú contextual ocurre cuando el usuario oprime y suelta el botón derecho del ratón. El método `isPopupTrigger` de `MouseEvent` devuelve `true` si ocurrió el evento de desencadenamiento del menú contextual.
- El método `show` de `JPopupMenu` muestra un objeto `JPopupMenu`. El primer argumento especifica el componente de origen, que ayuda a determinar en dónde aparecerá el objeto `JPopupMenu`. Los últimos dos argumentos son las coordenadas de la esquina superior izquierda del componente de origen, en donde aparece el objeto `JPopupMenu`.
- La clase `UIManager` contiene la clase anidada `LookAndFeelInfo` que mantiene la información acerca de una apariencia visual. El método estático `getInstalledLookAndFeels` de `UIManager` obtiene un arreglo de `UIManager.LookAndFeelInfo`, los cuales describen las apariencias visuales disponibles. El método estático `setLookAndFeel` de `UIManager` cambia la apariencia visual. El método estático `updateComponentTreeUI` de `SwingUtilities` cambia la apariencia visual de todos los componentes adjuntos a su argumento `Component`, a la nueva apariencia visual.
- Muchas de las aplicaciones de hoy en día utilizan una MDI (interfaz de múltiples documentos) para administrar varios documentos que se procesan en paralelo. Las clases `JDesktopPane` y `JInternalFrame` de Swing proporcionan soporte para crear interfaces de múltiples documentos.
- Un objeto `JTabbedPane` ordena los componentes de la GUI en capas, en donde sólo una capa está visible en un momento dado. Los usuarios acceden a cada una de las capas mediante una ficha; algo muy parecido a las carpetas en un archivero. Cuando los usuarios hacen clic en una ficha, se muestra la capa apropiada.
- `BoxLayout` es un administrador de esquemas que permite ordenar los componentes de la GUI de izquierda a derecha, o de arriba hacia abajo, en un contenedor. La clase `Box` declara un contenedor con `BoxLayout` como su administrador de esquemas predeterminado, y proporciona métodos estáticos para crear un objeto `Box` con un esquema `BoxLayout` horizontal o vertical.
- `GridBagLayout` es un administrador de esquemas similar a `GridLayout`. A diferencia de `GridLayout`, el tamaño de cada componente puede variar y pueden agregarse componentes en cualquier orden. Un objeto `GridBagConstraints` especifica cómo se coloca un componente en un esquema `GridBagLayout`. El método `setConstraints` de la clase `GridBagLayout` toma un argumento `Component` y un argumento `GridBagConstraints`, y establece las restricciones del objeto `Component`.

TERMINOLOGÍA

`addSeparator`, método de la clase `JMenu`
`addWindowListener`, método de la clase `Window`
ajustar a las marcas
apariencia visual de Mac
apariencia visual metálica
apariencia visual Motif
apariencia visual Windows
área dedicada de dibujo
argumentos de línea de comandos
barra de menús
`Box`, clase
`BoxLayout`, administrador de esquemas
`createHorizontalBox`, método de `Box`
`createHorizontalGlue`, método de `Box`
`createHorizontalStrut`, método de `Box`
`createRigidArea`, método de `Box`
`createVerticalBox`, método de `Box`
`createVerticalGlue`, método de `Box`
`createVerticalStrut`, método de `Box`
cuadro de diálogo modal
`ChangeEvent`, clase
`ChangeListener`, interfaz
`Dimension`, clase
`dispose`, método de la clase `Window`
elemento de menú
envoltura de líneas
etiquetas para marcas
evento externo
`first`, método de `GridLayout`
`getClassName`, método de `UIManager.LookAndFeelInfo`
`getInstalledLookAndFeel`, método de `UIManager`
`getMinimumSize`, método de `Component`
`getPreferredSize`, método de `Component`
`getSelectedText`, método de `JTextComponent`
`getValue`, método de la clase `JSlider`
`GridBagConstraints`, clase
`GridBagLayout`, administrador de esquemas
indicador de un objeto `JSlider`
`isPopupTrigger`, método de `MouseEvent`
`JCheckBoxMenuItem`, clase
`JDesktopPane`, clase
`JInternalFrame`, clase
`JMenu`, clase
`JMenuBar`, clase
`JMenuItem`, clase
`JPopupMenu`, clase
`JRadioButtonMenuItem`, clase
`JSlider`, clase
`JTextArea`, clase
`JTextComponent`, clase
marca
marca más distanciada
marca menos distanciada
MDI (interfaz de múltiples documentos)
menú
menú contextual sensible al contexto
método `setMnemonic` de `AbstractButton`
montante vertical

http://libros y soluciones.net

nemónico
`paintComponent`, método de `JComponent`
PLAF (apariencia visual adaptable)
políticas de barra de desplazamiento para un objeto `JScrollPane`
`setConstraints`, método
`setDefaultCloseOperation`, método
`setHorizontalScrollBarPolicy`, método
`setJMenuBar`, método
`setLineWrap`, método
`setLookAndFeel`, método
`setMajorTickSpacing`, método
`setOpaque`, método de la clase `JComponent`
`setPaintTicks`, método de la clase `JSlider`
`setSelected`, método de `AbstractButton`
`setTitle`, método de la clase `Frame`
`setVerticalScrollBarPolicy`, método
`show`, método de la clase `JPopupMenu`
submenú
`UIManager`, clase
`UIManager.LookAndFeelInfo`, clase
`updateComponentTreeUI`, método
ventana hija
ventana padre
`WindowConstants`
`WindowListener`, interfaz

EJERCICIOS DE AUTOEVALUACIÓN

14.1 Complete las siguientes oraciones:
a) La clase _____ se utiliza para crear un objeto menú.
b) El método _____ coloca una barra separadora en un menú.
c) Si se pasa `false` al método _____ de un objeto `TextArea`, se evita que su texto pueda ser modificado por el usuario.
d) Los eventos de `JSlider` son manejados por el método _____ de la interfaz _____.
e) La variable de instancia _____ de `GridBagConstraints` se establece en `CENTER` de manera predeterminada.

14.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Cuando el programador crea un objeto `JFrame`, como mínimo debe crearse y agregarse un menú al objeto `JFrame`.
b) La variable `fill` pertenece a la clase `GridBagLayout`.
c) La acción de dibujar en un componente de la GUI se realiza con respecto a la coordenada (0, 0) de la esquina superior izquierda del componente.
d) El texto de un objeto `JTextArea` siempre es de sólo lectura.
e) La clase `JTextArea` es una subclase directa de la clase `Component`.
f) El esquema predeterminado para un objeto `Box` es `BoxLayout`.

14.3 Encuentre el (los) error(es) en cada una de las siguientes instrucciones y explique cómo corregirlo(s).
a)
```
JMenubar b;
```
b)
```
miSlider = JSlider( 1000, 222, 100, 450 );
```
c)
```
gbc.fill = GridBagConstraints.NORTHWEST; // establecer fill
```
d)
```
// sobrescribir a paint en un componente de Swing personalizado
public void paintcomponent( Graphics g )
{
   g.drawString( "HOLA" , 50 , 50 );
}
```
e)
```
// crear un objeto JFrame y mostrarlo
JFrame f = new JFrame( "Una ventana" );
f.setVisible( true );
```

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

14.1 a) `JMenu`. b) `addSeparator`. c) `setEditable`. d) `stateChanged`, `ChangeListener`. e) `anchor`.

14.2 a) Falso. Un objeto `JFrame` no requiere menús.
b) Falso. La variable `fill` pertenece a la clase `GridBagConstraints`.
c) Verdadero.
d) Falso. Los objetos `JTextArea` pueden editarse de manera predeterminada.
e) Falso. `JTextArea` se deriva de la clase `JTextComponent`.
f) Verdadero.

14.3 a) `JMenubar` debe ser `JMenuBar`.
b) El primer argumento para el constructor debe ser `SwingConstants.HORIZONTAL` o `SwingConstants.VERTICAL`, y debe utilizarse la palabra clave `new` después del operador =.
c) La constante debe ser `BOTH`, `HORIZONTAL`, `VERTICAL` o `NONE`.
d) `paintcomponent` debe ser `paintComponent`, y el método debe llamar a `super.paintComponent( g )` como su primera instrucción.
e) El método `setSize` de `JFrame` debe ser llamado también, para establecer el tamaño de la ventana.

EJERCICIOS

14.4 Complete las siguientes oraciones:
a) Un área dedicada de dibujo puede declararse como una subclase de _____.
b) Un objeto `JMenuItem` que es un `JMenu` se llama _____.
c) Tanto los objetos `JTextField` como los objetos `JTextArea` extienden directamente a la clase _____.
d) El método _____ adjunta un objeto `JMenuBar` a un objeto `JFrame`.
e) La clase contenedora _____ tiene un esquema `BoxLayout` predeterminado.
f) Un _____ administra a un conjunto de ventanas hijas declaradas con la clase `JInternalFrame`.

14.5 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Los menús requieren un objeto `JMenuBar` para poder adjuntarse a un objeto `JFrame`.
b) Un objeto `JPanel` es capaz de recibir eventos de ratón.
c) `BoxLayout` es el administrador de esquemas predeterminado para un objeto `JFrame`.
d) El método `setEditable` es un método de `JTextComponent`.
e) Los objetos `JPanel` son contenedores, a los cuales pueden adjuntarse otros componentes de la GUI.
f) La clase `JFrame` extiende directamente a la clase `Container`.
g) Los objetos `JApplet` pueden contener menús.

14.6 Encuentre el (los) error(es) en cada una de las siguientes instrucciones. Explique cómo corregir el (los) error(es).
a)
```
x.add( new JMenuItem( "Submenú Color" ) ); // crear submenú
```
b)
```
contenedor.setLayout( m = new GridbagLayout() );
```
c)
```
String s = JTextArea.getText();
```

14.7 Escriba un programa que muestre un círculo de tamaño aleatorio, calcule y muestre su área, radio, diámetro y circunferencia. Use las siguientes ecuaciones: *diametro* = 2 ∞ *radio*, *area* = π ∞ *radio*^{2}, *circunferencia* = 2 ∞ π ∞ *radio*. Use la constante `MathPI` para pi (π). Todos los dibujos deberán realizarse en una subclase de `JPanel`, y los resultados de los cálculos deberán mostrarse en un objeto `JTextArea` de sólo lectura.

14.8 Mejore el programa del ejercicio 14.7 al permitir al usuario alterar el radio con un objeto `JSlider`. El programa deberá funcionar para todos los radios en el rango de 100 a 200. A medida que cambie el radio, el diámetro, área y circunferencias deberán actualizarse y mostrarse. El radio inicial debe ser `150`. Use las ecuaciones del ejercicio 14.7. Todos los dibujos deberán realizarse en una subclase de `JPanel`, y los resultados de los cálculos deberán mostrarse en un objeto `JTextArea` de sólo lectura.

14.9 Explore los efectos de variar los valores de `weightx` y `weighty` del programa de la figura 14.19. ¿Qué ocurre cuando una ranura tiene valores distintos de cero para `weightx` y `weighty`, pero no se le permite rellenar toda el área (es decir, que el valor `fill` no sea `BOTH`)?

14.10 Escriba un programa que utilice el método `paintComponent` para dibujar el valor actual de un objeto `JSlider` en una subclase de `JPanel`. Además, proporcione un objeto `JTextField` en donde pueda introducirse un valor específico. El objeto `JTextField` deberá mostrar el valor actual del objeto `JSlider` en todo momento. Debe usarse un objeto `JLabel` para identificar al objeto `JTextField`. Deben utilizarse los métodos `setValue` y `getValue` de `JSlider`. [*Nota:* El método `setValue` es un método `public` que no devuelve un valor y toma un argumento entero: el valor de `JSlider`, que determina la posición del indicador.]

14.11 Modifique el programa de la figura 14.13, agregando un mínimo de dos fichas nuevas.

14.12 Declare a una subclase de `JPanel` llamada `MiSelectorDeColores`, que proporcione tres objetos `JSlider` y tres objetos `JTextField`. Cada objeto `JSlider` representa los valores de 0 a 255 para las partes rojo, azul y verde de un color. Use los valores de rojo, verde y azul como argumentos para el constructor de `Color`, para crear un nuevo objeto `Color`. Muestre el valor actual de cada objeto `JSlider` en el objeto `JTextField` correspondiente. Cuando el usuario cambie el valor del objeto `JSlider`, el objeto `JTextField` deberá cambiar de manera acorde. Declare la clase `MiSelectorDeColores` de manera que pueda reutilizarse en otras aplicaciones o subprogramas (applets). Use su nuevo componente de la GUI como parte de un subprograma que muestre el valor actual de `Color`, dibujando un rectángulo relleno.

http://librosysolucionarios.net

14.13 Modifique la clase `MiSelectorDeColor` del ejercicio 14.12 para permitir al usuario introducir un valor entero en un objeto `JTextField`, para establecer el valor rojo, verde o azul. Cuando el usuario oprima *Intro* en el objeto `JTextField`, deberá establecerse el objeto `JSlider` correspondiente en el valor apropiado.

14.14 Modifique el subprograma del ejercicio 14.13 para dibujar el color actual como un rectángulo en una instancia de una subclase de `JPanel`, llamada `PanelDibujo`. La clase `PanelDibujo` debe proporcionar su propio método `paintComponent` para dibujar el rectángulo, y debe proporcionar métodos *establecer* para establecer los valores de rojo, verde y azul para el color actual. Cuando se invoque a uno de los métodos *establecer* para la clase `PanelDibujo`, el objeto deberá repintarse automáticamente a sí mismo (mediante `repaint`).

14.15 Modifique el subprograma del ejercicio 14.14 para permitir al usuario arrastrar el ratón a lo largo del `PanelDibujo`, para dibujar una figura con el color actual. Permita al usuario seleccionar la figura a dibujar.

14.16 Modifique el programa del ejercicio 14.15 para permitirle ejecutarse como una aplicación. El código existente del subprograma debe modificarse solamente agregando un método `main` para iniciar la aplicación en su propio objeto `JFrame`. Proporcione al usuario la habilidad de terminar la aplicación, haciendo clic en el cuadro para cerrar la ventana y seleccionando `Salir` de un menú `Archivo`. Use las técnicas que se muestran en la figura 14.9.

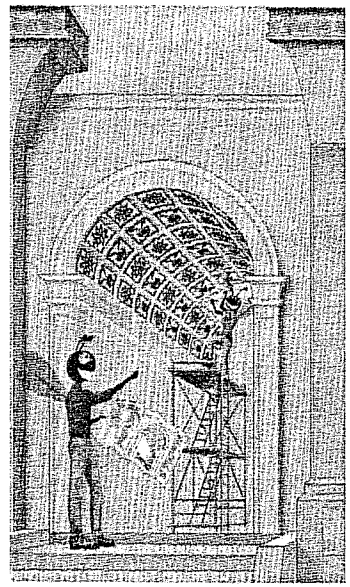
14.17 *(Aplicación de dibujo completa.)* Utilizando las técnicas desarrolladas en los ejercicios 13.27 a 13.31 y en los ejercicios 14.12 a 14.16, cree un programa de dibujo completo que pueda ejecutarse como subprograma y como aplicación. El programa debe utilizar los componentes de la GUI de los capítulos 13 y 14, para permitir al usuario seleccionar la figura, color y características de relleno. Cada figura deberá guardarse en un arreglo de objetos `MiFigura`, en donde `MiFigura` es la superclase en su jerarquía de clases de figuras (vea los ejercicios 10.9 y 10.10). Use un objeto `JDesktopPane` y objetos `JInternalFrame` para permitir al usuario crear varios dibujos separados, en ventanas hijas separadas. Cree la interfaz de usuario como una ventana hija separada que contenga todos los componentes de la GUI que permitan al usuario determinar las características de la figura a dibujar. El usuario podrá entonces hacer clic en cualquier objeto `JInternalFrame` para dibujar la figura.

15

Manejo de excepciones

Objetivos

- Comprender el manejo de excepciones y de errores.
- Utilizar `try`, `throw` y `catch` para detectar, indicar y manejar excepciones, respectivamente.
- Utilizar la cláusula `finally` para liberar recursos.
- Comprender la jerarquía de excepciones de Java.
- Declarar nuevas clases de excepciones.
- Crear excepciones encadenadas.



Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.
Franklin Delano Roosevelt

¡Oh! Arroja la peor parte de ello,
y vive en forma más pura con la otra mitad.
William Shakespeare

Si están corriendo y no saben hacia dónde se dirigen
tengo que salir de alguna parte y atraparlos.
Jerome David Salinger

A menudo, la excusa de una falta hace que ésta
sea aún peor debido a la excusa.
William Shakespeare

Nunca olvido un rostro, pero en su caso haré una excepción.
Groucho (Julius Henry) Marx

http://librosysolucionarios.net

Plan general

15.1 Introducción
15.2 Generalidades acerca del manejo de excepciones
15.3 Ejemplo de manejo de excepciones: División entre cero
15.4 Jerarquía de excepciones en Java
15.5 Volver a lanzar una excepción
15.6 Cláusula `finally`
15.7 Limpieza de la pila
15.8 `printStackTrace`, `getStackTrace` y `getMessage`
15.9 Excepciones encadenadas
15.10 Declaración de nuevos tipos de excepciones
15.11 Constructores y manejo de excepciones
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

15.1 Introducción

En este capítulo presentaremos el *manejo de excepciones*. Una *excepción* es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre "excepción" viene del hecho de que, aunque puede ocurrir un problema, éste ocurre con poca frecuencia; si la "regla" es que una instrucción generalmente se ejecuta en forma correcta, entonces la "excepción a la regla" es cuando ocurre un problema. El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubiera encontrado el problema. Un problema más grave podría evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema antes de terminar de una manera controlada. Las características que presentamos en este capítulo permiten a los programadores escribir *programas tolerantes a fallas* y robustos. El estilo y los detalles sobre el manejo de excepciones en Java se basan, en parte, en el trabajo que Andrew Koenig y Bjarne Stroustrup presentaron en su artículo "Exception Handling for C++ (versión revisada)."[1]

El capítulo empieza con un panorama general de los conceptos relacionados con el manejo de excepciones, y posteriormente se demuestran las técnicas básicas para el manejo de excepciones. Mostraremos estas técnicas mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando un método intenta realizar una división entre cero. Después del ejemplo, presentaremos la jerarquía de excepciones en Java, la cual incluye varias clases que proporcionan los paquetes de Java para el manejo de excepciones. Luego hablaremos sobre la nueva característica de excepciones encadenada, la cual se incluye en la Plataforma Java 2, Standard Edition, versión 1.4. Concluiremos este capítulo hablando sobre cuestiones adicionales acerca del manejo de excepciones, como la manera de manejar excepciones que ocurren en un constructor o finalizador.

15.2 Generalidades acerca del manejo de excepciones

La lógica del programa evalúa frecuentemente condiciones que determinan cómo debe proceder la ejecución del programa. Considere el siguiente seudocódigo:

1. Koenig, A. y B. Stroustrup. "Exception Handling for C++ (versión revisada)," *Proceedings of the USENIX C++ Conference*, pp. 149-176, San Francisco, Abril de 1990.

http://librosysolucionarios.net


```
Realizar una tarea

Si la tarea anterior no se ejecutó correctamente
    Realizar el procesamiento de los errores

Realizar la siguiente tarea

Si la tarea anterior no se ejecutó correctamente
    Realizar el procesamiento de los errores

...
```

En este seudocódigo empezamos realizando una tarea; después, evaluamos si esa tarea se ejecutó correctamente. Si no lo hizo, realizamos el procesamiento de los errores. De otra manera, continuamos con la siguiente tarea. Aunque esta forma de manejo de errores funciona, al entremezclar la lógica del programa con la lógica del manejo de errores el programa podría ser difícil de leer, mantener y depurar; especialmente en aplicaciones extensas. De hecho, si los problemas potenciales ocurren con poca frecuencia, al entremezclar la lógica del programa y la lógica del manejo de errores se puede degradar el rendimiento del programa, ya que éste debe evaluar la lógica de manejo de errores para determinar si se puede llevar a cabo la siguiente tarea. El manejo de excepciones permite al programador remover el código para manejo de errores de la "línea principal" de ejecución del programa, lo cual mejora la claridad y capacidad de modificación del mismo. Los programadores pueden optar por manejar las excepciones que elijan: todas las excepciones, todas las excepciones de cierto tipo o todas las excepciones de un grupo de tipos relacionados (por ejemplo, los tipos de excepciones que pertenecen a una jerarquía de herencia). Esta flexibilidad reduce la probabilidad de que los errores se pasen por alto y, por consecuencia, hace que un programa sea más robusto.

Tip para prevenir errores 15.1



El manejo de excepciones ayuda a mejorar el grado de tolerancia a fallas de un programa.

Observación de ingeniería de software 15.1



Evite usar el manejo de excepciones para manipular los problemas potenciales en el flujo de control convencional. Manejar un número extenso de casos de excepciones puede ser incómodo; además, los programas con un número extenso de casos de excepciones pueden ser difíciles de leer y de mantener.

Observación de ingeniería de software 15.2



Otra de las razones por las que no se deben usar las excepciones para el flujo de control convencional es que estas excepciones "adicionales" pueden "interponerse en el camino" de las excepciones de tipos de errores genuinos. Es más difícil para el programador llevar el registro de un número más extenso de casos de excepciones. Las situaciones excepcionales deben ser raras, no comunes.

El manejo de excepciones está diseñado para procesar *errores síncrónicos*, que ocurren cuando se ejecuta una instrucción. Ejemplos comunes de estos errores son los índices fuera de rango, el desbordamiento aritmético (es decir, un valor fuera del rango representable de valores), la división entre cero, los parámetros inválidos de método, la interrupción de subprocesos y la asignación fallida de memoria (debido a la falta de ésta). El manejo de excepciones no está diseñado para procesar los problemas asociados con los eventos *asíncronos* (por ejemplo, completar las operaciones de E/S de disco, la llegada de mensajes de red, clics del ratón y pulsaciones de teclas), los cuales ocurren en paralelo con, y en forma independiente de, el flujo de control del programa.

Buena práctica de programación 15.1



Por cuestión de claridad, evite el uso del manejo de excepciones para otros propósitos distintos del manejo de los errores que ocurren en un programa.

Con lenguajes de programación que no soportan el manejo de excepciones, los programadores a menudo retrasan la escritura de código de procesamiento de errores, o algunas veces olvidan incluirlo. Esto hace que los productos de software sean menos robustos. Java permite al programador tratar con el manejo de excepciones fácilmente, desde el comienzo de un proyecto. Sin embargo, el programador debe incorporar una estrategia de manejo de excepciones en los proyectos de software.

Observación de ingeniería de software 15.3



Trate de incorporar una estrategia de manejo de excepciones en un sistema, desde el comienzo del proceso de diseño. Puede ser difícil incluir un manejo efectivo de excepciones una vez que se haya implementado el sistema.

Observación de ingeniería de software 15.4



En el pasado, los programadores han utilizado muchas técnicas para implementar el código para procesar errores. El manejo de excepciones proporciona una sola técnica uniforme para el procesamiento de los problemas. Esto ayuda a que cada uno de los programadores que trabajan en proyectos grandes comprenda el código de procesamiento de errores de los otros.

El mecanismo de manejo de excepciones también es útil para procesar los problemas que ocurren cuando un programa invoca a los métodos de otras clases. En vez de manejar los problemas internamente, dichos métodos utilizan por lo común excepciones para notificar a los métodos que hacen las llamadas cuándo ocurren los problemas. Esto permite a los programadores implementar un manejo de errores personalizado para cada aplicación.

Tip de rendimiento 15.1



Cuando no ocurren excepciones, el código para manejar excepciones afecta muy poco, o nada, al rendimiento del programa. Debido a esto, los programas que implementan el manejo de excepciones operan con mucha más eficiencia que los programas que entremezclan el código para manejo de errores con la lógica del programa.

Observación de ingeniería de software 15.5



Es más conveniente que los métodos con condiciones de error comunes devuelvan `false` o `null` (o cualquier otro valor apropiado), en vez de lanzar excepciones. Un programa que llame a dicho método puede comprobar el valor de retorno para determinar el éxito o el fracaso de la llamada al método.

Las aplicaciones complejas normalmente consisten de componentes predefinidos de software y de componentes específicos para cada aplicación. Cuando un componente predefinido se encuentra un problema, ese componente necesita de un mecanismo para comunicar el problema al componente específico de la aplicación; el componente predefinido no puede saber de antemano cómo procesa cada aplicación cierto problema ocurrido. El manejo de excepciones simplifica la combinación de componentes de software y les permite trabajar efectivamente, al permitir que los componentes predefinidos comuniquen los problemas a los componentes específicos de la aplicación, los cuales podrán entonces procesar los problemas de una manera específica para cada aplicación.

El manejo de excepciones está dirigido a situaciones en las que el método que detecta un problema es incapaz de manejarlo. Dicho método *lanza una excepción*. No hay garantía de que habrá un *manejador de excepciones* (código que se ejecuta cuando el programa detecta una excepción) para procesar ese tipo de excepción. Si existe, el manejador de excepciones *atrapa* y *maneja* a esa excepción. El resultado de una *excepción no atrapada* a menudo produce efectos adversos y podría terminar con la ejecución del programa.

Error común de programación 15.1



Si se aborta el componente de un programa debido a una excepción no atrapada, se podría dejar un recurso (como un flujo de archivos o un dispositivo de E/S) en un estado en el que los demás programas no puedan adquirirlo. A esto se le conoce como "fuga de recursos".

Java proporciona las *instrucciones* `try` para permitir el manejo de excepciones. Una instrucción `try` consiste de la palabra clave `try`, seguida por llaves ({}) que delimitan a ese *bloque* `try`. El bloque `try` contiene instrucciones que podrían ocasionar excepciones, e instrucciones que no deberían ejecutarse en caso de que ocurra una instrucción. Debe haber por lo menos una *cláusula* `catch` (a la que también se le llama manejador de excepciones) o una *cláusula* `finally` inmediatamente después del bloque `try`. Cada cláusula `catch` especifica entre paréntesis un *parámetro de excepción*, el cual identifica al tipo de excepción que puede procesar el manejador. El nombre del parámetro de excepción permite que la cláusula `catch` interactúe con un objeto de excepción atrapada. Después del último manejador `catch`, una cláusula `finally` opcional proporciona código que siempre se ejecuta, sin importar que ocurra o no una excepción. Más adelante veremos que la cláusula `finally` es una ubicación ideal para el código que libera recursos, para evitar las "fugas de recursos".

http://librosysolucionarios.net

Error común de programación 15.2



Es un error de sintaxis colocar código entre un bloque `try` y sus correspondientes cláusulas `catch`.

Error común de programación 15.3



Especificar una lista de parámetros de `catch` separada por comas es un error de sintaxis. Una instrucción `catch` sólo puede tener un parámetro.

Error común de programación 15.4



Es un error de compilación atrapar el mismo tipo en dos cláusulas `catch` distintas, asociadas con un bloque `try` específico.

Error común de programación 15.5



Si se colocara una cláusula `catch` para el tipo `Exception` antes de que otras cláusulas `catch` que atrapen tipos de excepciones de subclases, podría evitarse que esas cláusulas se ejecuten, con lo que se produciría un error de compilación.

El punto en el programa en el que ocurre una excepción (es decir, la ubicación en la que un método detecta y lanza una excepción) se conoce como el *punto de lanzamiento*. Si ocurre una excepción en un bloque `try`, ese bloque termina inmediatamente y el control del programa se transfiere a la primera cláusula `catch` que vaya después del bloque `try`. Esto se conoce como *modelo de terminación del manejo de excepciones*, ya que el bloque `try` que encierra a una excepción lanzada termina al ocurrir esa excepción[2]. Al igual que con cualquier otro bloque de código, cuando termina un bloque `try` las variables locales que estén declaradas en el bloque quedan fuera de alcance. A continuación, el programa busca la primera cláusula `catch` que pueda procesar el tipo de excepción que ocurrió. El programa ubica la cláusula `catch` que concuerde, comparando el tipo de la excepción lanzada con el tipo del parámetro de excepción de cada cláusula `catch`, hasta que el programa encuentre una concordancia. La concordancia ocurre si los tipos son idénticos, o si el tipo de la excepción lanzada es una subclase del tipo del parámetro de excepción. Cuando ocurre una concordancia, se ejecuta el código contenido dentro del manejador `catch` concordante. Cuando una cláusula `catch` termina de procesarse, las variables locales que se declaran dentro de la cláusula (incluyendo su parámetro) quedan fuera de alcance. Cualquier cláusula `catch` restante que corresponda a ese bloque `try` se ignora, y la ejecución continúa en la primera línea de código después de la secuencia `try/catch`.

Observación de ingeniería de software 15.6



Si usted sabe que un método podría lanzar una excepción, incluya el código apropiado para manejar excepciones en su programa. Esto hará que su programa sea más robusto.

Buena práctica de programación 15.2



Lea la documentación en línea de la API para obtener información sobre un método, antes de utilizarlo en un programa. La documentación especifica las excepciones lanzadas por el método (en caso de haberlas) e indica las razones por las que podrían ocurrir dichas excepciones.

Buena práctica de programación 15.3



Lea la documentación en línea de la API para obtener información sobre una clase de excepción, antes de escribir código para manejar ese tipo de excepción. La documentación para una clase de excepción generalmente contiene las razones potenciales por las que dichas excepciones podrían ocurrir durante la ejecución de un programa.

Si no ocurren excepciones en un bloque `try`, el programa ignora el (los) manejador(es) para ese bloque. La ejecución del programa continúa con la siguiente instrucción que haya después de la secuencia `try/catch`. Si aparece una cláusula `finally` después de la última cláusula `catch`, la cláusula `finally` se ejecutará sin importar que ocurra o no una excepción. Si ocurre una excepción en un método y no es atrapada, o si la instrucción que produjo la excepción no se encuentra dentro de un bloque `try`, el método que contiene la instrucción termina inmediatamente y el programa trata de localizar un bloque `try` circundante en el método que hizo la llamada. (Este proceso, que se describe en la sección 15.7, se conoce como *limpieza de la pila*.)

[2] Algunos lenguajes utilizan el *modelo de reanudación del manejo de excepciones* en el que, después de manejar la excepción, el control regresa al punto en el que se lanzó la excepción y la ejecución se reanuda a partir de ese punto.

En la declaración de un método, una *cláusula* **`throws`** especifica las excepciones que lanza ese método. Esta cláusula aparece después de la lista de parámetros y antes del cuerpo del método. La cláusula contiene una lista separada por comas de las excepciones que lanzará el método, si ocurre un problema cuando éste se ejecute. Dichas excepciones pueden ser lanzadas por instrucciones en el cuerpo del método, o pueden lanzarse mediante los métodos que se llamen en el cuerpo. Un método puede lanzar excepciones de clases indicadas, o puede lanzar excepciones de sus subclases.

15.3 Ejemplo de manejo de excepciones: División entre cero

Considere un simple ejemplo del manejo de excepciones. La aplicación de la figura 15.1 utiliza **`try`** y **`catch`** para envolver código que podría lanzar una excepción de "división entre cero" y, para manejar esa excepción, debe ocurrir una. La aplicación muestra dos objetos **`JTextField`** en los que el usuario puede introducir enteros. Cuando el usuario oprime la tecla *Intro* en el segundo objeto **`JTextField`**, el programa llama al método **`actionPerformed`** para leer los dos enteros de los objetos **`JTextField`** y pasa los enteros al método **`cociente`**, el cual calcula el cociente de los dos valores y devuelve un resultado **`int`**. Java no permite la división entre cero, en la aritmética con enteros. Cuando esto ocurre, Java lanza una excepción **`ArithmeticException`**. En este ejemplo, si el usuario escribe 0 en el segundo objeto **`JTextField`**, se produce una excepción **`ArithmeticException`** y el programa atrapa a esa excepción. Además, en los ejemplos anteriores en los que se leyeron valores numéricos del usuario, se suponía que éste escribiría un valor entero apropiado. Sin embargo, los usuarios algunas veces cometen errores e introducen valores no enteros. Si el método **`parseInt`** de la clase **`Integer`** recibe una cadena que no representa a un entero válido, el método lanza una excepción **`NumberFormatException`**. Por lo tanto, este programa demuestra también cómo atrapar excepciones **`NumberFormatException`**.

```
1  // Fig. 15.1: PruebaDivisionEntreCero.java
2  // Un ejemplo de manejo de excepciones que comprueba la división entre cero.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PruebaDivisionEntreCero extends JFrame
8     implements ActionListener {
9
10    private JTextField campoEntrada1, campoEntrada2, campoSalida;
11    private int numero1, numero2, resultado;
12
13    // configurar la GUI
14    public PruebaDivisionEntreCero()
15    {
16       super( "Demostración de las excepciones" );
17
18       // obtener panel de contenido y establecer su esquema
19       Container contenedor = getContentPane();
20       contenedor.setLayout( new GridLayout( 3, 2 ) );
21
22       // establecer etiqueta y campoEntrada1
23       contenedor.add(
24          new JLabel( "Escriba el numerador ", SwingConstants.RIGHT ) );
25       campoEntrada1 = new JTextField();
26       contenedor.add( campoEntrada1 );
27
28       // establecer etiqueta y campoEntrada2; registrar componente de escucha
```

Figura 15.1 Ejemplo de manejo de excepciones con división entre cero. (Parte 1 de 3.)


```
29       contenedor.add( new JLabel( "Escriba el denominador y oprima Intro ",
30          SwingConstants.RIGHT ) );
31       campoEntrada2 = new JTextField();
32       contenedor.add( campoEntrada2 );
33       campoEntrada2.addActionListener( this );
34
35       // establecer etiqueta y campoSalida
36       contenedor.add( new JLabel( "RESULTADO ", SwingConstants.RIGHT ) );
37       campoSalida = new JTextField();
38       contenedor.add( campoSalida );
39
40       setSize( 425, 100 );
41       setVisible( true );
42
43    } // fin del constructor de PruebaDivisionEntreCero
44
45    // procesar eventos de la GUI
46    public void actionPerformed( ActionEvent evento )
47    {
48       campoSalida.setText( "" );   // borrar campoSalida
49
50       // leer dos números y calcular el cociente
51       try {
52          numero1 = Integer.parseInt( campoEntrada1.getText() );
53          numero2 = Integer.parseInt( campoEntrada2.getText() );
54
55          resultado = cociente( numero1, numero2 );
56          campoSalida.setText( String.valueOf( resultado ) );
57       }
58
59       // procesar la entrada con formato incorrecto
60       catch ( NumberFormatException excepcionFormatoNumero ) {
61          JOptionPane.showMessageDialog( this,
62             "Debe escribir dos enteros", "Formato de número inválido",
63             JOptionPane.ERROR_MESSAGE );
64       }
65
66       // procesar los intentos de dividir entre cero
67       catch ( ArithmeticException excepcionAritmetica ) {
68          JOptionPane.showMessageDialog( this,
69             excepcionAritmetica.toString(), "Excepción aritmética",
70             JOptionPane.ERROR_MESSAGE );
71       }
72
73    } // fin del método actionPerformed
74
75    // demuestra cómo lanzar una excepción cuando ocurre una división entre cero
76    public int cociente( int numerador, int denominador )
77       throws ArithmeticException
78    {
79       return numerador / denominador;
80    }
81
82    public static void main( String args[] )
```

http://librosysolucionarios.net

Figura 15.1 Ejemplo de manejo de excepciones con división entre cero. (Parte 2 de 3.)

```
83    {
84       PruebaDivisionEntreCero aplicacion = new PruebaDivisionEntreCero();
85       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
86    }
87
88 } // fin de la clase PruebaDivisionEntreCero
```

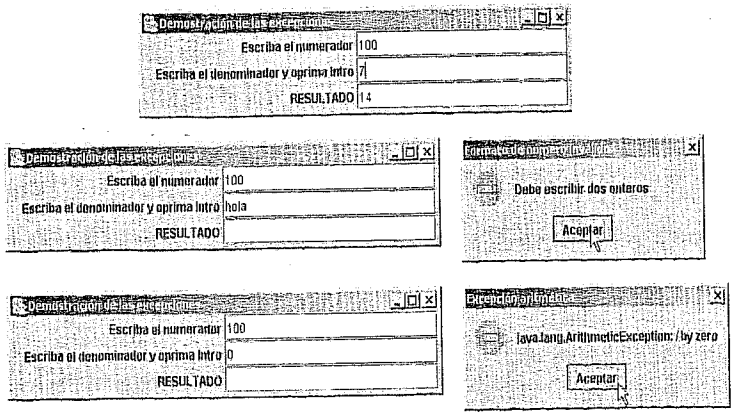


Figura 15.1 Ejemplo de manejo de excepciones con división entre cero. (Parte 3 de 3.)

En la salida de la figura 15.1, la primera ventana muestra una división exitosa. En las siguientes dos ventanas de resultados, el usuario introdujo la cadena `"hola"` en el segundo objeto `JTextField`. Cuando el usuario oprime *Intro* en el segundo objeto `JTextField` se muestra un cuadro de diálogo de mensaje de error, indicando que debe introducirse un entero. En las últimas dos ventanas, el usuario introdujo un denominador de cero. El programa detectó el problema, lanzó una excepción y emitió un mensaje de diagnóstico apropiado.

Ahora considere la aplicación `PruebaDivisionEntreCero` (figura 15.1). El constructor de la aplicación (líneas 14 a 43) crea una interfaz gráfica de usuario con tres objetos `JLabel` (todos con alineación a la derecha) y tres objetos `JTextField`, y registra el objeto `PruebaDivisionEntreCero` como el objeto `ActionListener` para el objeto `JTextField` `campoEntrada2` (línea 33).

Cuando el usuario introduce el denominador y oprime *Intro*, el programa llama al método `actionPerformed` (líneas 46 a 73), el cual empieza por borrar el contenido de `campoSalida` (línea 48). A continuación, el método `actionPerformed` continúa con un bloque `try` (líneas 51 a 57), el cual encierra el código que podría lanzar (mediante `throw`) una excepción, y el código que no debe ejecutarse si ocurre una excepción. Las instrucciones que leen los enteros de los objetos `JTextField` (líneas 52 y 53) utilizan al método `Integer.parseInt` para convertir una cadena en un valor `int`. El método `parseInt` lanza una excepción `NumberFormatException` si su argumento no representa a un entero válido. La división que puede ocasionar una excepción `ArithmeticException` no se lleva a cabo en forma explícita, en el bloque `try`. En vez de ello, la llamada al método `cociente` (línea 55) contiene el código que intenta realizar la división. El método `cociente` (líneas 76 a 80) lanza el objeto `ArithmeticException`, como veremos en unos momentos. En general, las excepciones pueden surgir a través del código mencionado en forma explícita en un bloque `try`, mediante llamadas a otros métodos o incluso a través de llamadas a métodos profundamente anidadas, iniciadas por el código en un bloque `try`.

El bloque `try` en este ejemplo va seguido de dos cláusulas `catch`: las líneas 60 a 64 contienen la cláusula `catch` para una excepción `NumberFormatException` y las líneas 67 a 71 contienen la cláusula `catch` para una excepción `ArithmeticException`. En general, cuando el programa detecta una excepción mientras ejecuta un bloque `try`, atrapa la excepción en una cláusula `catch` que especifica el manejador de excepción apropiado (es decir, el tipo en la cláusula `catch` concuerda exactamente con el tipo de excepción lanzada, o es una superclase del tipo de excepción lanzada). En la figura 15.1, la primera cláusula `catch` especifica que atrapará los objetos de excepción de tipo `NumberFormatException` (este tipo concuerda con el tipo de objeto de excepción lanzada en el método `Integer.parseInt`) y la segunda cláusula `catch` especifica que atrapará los objetos de excepción de tipo `ArithmeticException`. Sólo la cláusula `catch` que concuerde se ejecutará cuando ocurra una excepción. Ambas cláusulas `catch` simplemente muestran un cuadro de mensaje de error, pero las cláusulas `catch` pueden ser más elaboradas. Después de ejecutar una cláusula `catch`, el flujo de control de este programa continúa con la primera instrucción que está después de la última cláusula `catch`. En este caso, el control procede a la línea 72, y el método `actionPerformed` regresa al método que hizo la llamada.

Si el código en el bloque `try` no lanza (`throw`) una excepción, entonces se ignoran los manejadores `catch` que siguen después de ese bloque `try`, y la ejecución se reanuda en la primera línea de código que está después de los manejadores `catch`. (De nuevo, el método `actionPerformed` termina de ejecutarse). En la figura 15.1, el método `actionPerformed` simplemente regresa, pero el programa podría seguir ejecutando más instrucciones después de las cláusulas `catch`.

Error común de programación 15.6

Pueden ocurrir errores lógicos si usted supone que, después de procesar una excepción, el control regresará a la primera instrucción que sigue después de `throw`. *El control del programa continúa con la primera instrucción que sigue después de los manejadores* `catch`.

Tip para prevenir errores 15.2

Con el manejo de excepciones, un programa puede seguir ejecutándose después de hacerse cargo de un problema. Esto ayuda a asegurar que se desarrollen aplicaciones robustas, que contribuyan a lo que se conoce como computación de misión crítica, o computación crítica para los negocios.

Ahora examinemos el método `cociente` (líneas 76 a 80). Cuando el `denominador` es cero, Java lanza un objeto `ArithmethicException` para la división de la línea 79. Este objeto será atrapado por la cláusula `catch` (líneas 67 a 71) en `actionPerformed`. El manejador `ArithmeticException` convierte la excepción en una cadena mediante `toString`, y muestra esta cadena como el mensaje en un cuadro de diálogo de mensaje de error.

Si `denominador` no es cero, `cociente` realiza la división y devuelve el resultado de la división hasta el punto de la invocación del método `cociente` en el bloque `try` (línea 55). En la línea 56 se muestra el resultado del cálculo en el tercer objeto `JTextField`. En este caso, el bloque `try` se completa con éxito, por lo que el programa ignora las cláusulas `catch` y el método `actionPerformed` completa su ejecución en forma normal.

Observe que, cuando `cociente` lanza la excepción `ArithmeticException`, `cociente` termina y no devuelve un valor, lo cual hace que todas las variables locales de `cociente` queden fuera de alcance. Si `cociente` tuviera variables locales que fueran referencias a objetos, se reduciría de manera acorde el conteo de las referencias a esos objetos, y éstos posiblemente se marcarían para la recolección de basura. Además, cuando ocurre la excepción el bloque `try` (desde el cual se llamó a `cociente`) termina antes de que la línea 56 pueda ejecutarse. Aquí también, si fueran variables locales creadas en el bloque `try` antes de la excepción que se está lanzando, estas variables quedarían fuera de alcance.

Si se genera una excepción `NumberFormatException` mediante las líneas 52 y 53, el bloque `try` termina y la ejecución continúa con la cláusula `catch` de la línea 60, la cual muestra un mensaje de error para indicar al usuario que introduzca enteros. En este caso, nunca se hace la llamada al método `cociente`. Después el método `actionPerformed` continúa con la siguiente instrucción válida después de las cláusulas `catch` (en este ejemplo, el método termina).

15.4 Jerarquía de excepciones en Java

En esta sección examinaremos la jerarquía de herencia de las excepciones en Java. Las excepciones, al igual que casi todo lo demás en Java, son objetos. Por lo tanto, los programadores pueden crear jerarquías de clases de excepciones. En la figura 15.2 se muestra una pequeña porción de la jerarquía de herencia para la clase `Throwable` (una subclase de `Object`), la cual es la superclase de todas las excepciones. Sólo pueden usarse

http://librosysolucionarios.net

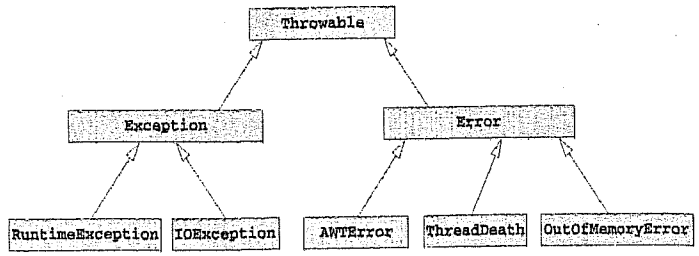


Figura 15.2 Jerarquía de herencia para la clase `Throwable`.

objetos `Throwable` con el mecanismo de manejo de excepciones. La clase `Throwable` tiene dos subclases: `Exception` y `Error`. La clase `Exception` y sus subclases (por ejemplo, `RuntimeException` e `IOException`, ambas del paquete `java.lang`) representan situaciones excepcionales que podrían ocurrir en un programa de Java, y que podrían ser atrapadas por la aplicación. La clase `Error` y sus subclases (por ejemplo, `OutOfMemoryError`) representan situaciones excepcionales que podrían ocurrir en el sistema en tiempo de ejecución en Java, pero que generalmente no deben ser atrapadas por una aplicación.

Java clasifica a las excepciones en dos categorías: *excepciones verificadas* y *excepciones no verificadas*. Esta distinción es importante, ya que el compilador de Java implementa un *requerimiento de atrapar o declarar* para las excepciones verificadas. El compilador *verifica* cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza excepciones verificadas. De ser así, el compilador asegura que la excepción verificada sea atrapada o declarada en una cláusula `throws`. Para satisfacer la parte relacionada con *atrapar* del requerimiento de atrapar o declarar, el código que genera la excepción debe envolverse en un bloque `try`, y debe proporcionar un manejador `catch` para el tipo de excepción verificada (o uno de los tipos de su superclase). Para satisfacer la parte relacionada con *declarar* del requerimiento de atrapar o declarar, el método que contiene el código que genera la excepción debe proporcionar una cláusula `throws` que contenga el tipo de excepción verificada, después de su lista de parámetros y antes de su cuerpo. Si el requerimiento de atrapar o declarar no se satisface, el compilador emitirá un mensaje de error, indicando que la excepción debe ser atrapada o declarada.



Error común de programación 15.7

Se producirá un error de compilación si un método intenta lanzar una excepción verificada que no esté listada en la cláusula `throws` de ese método.



Error común de programación 15.8

Si el método de una subclase sobrescribe al método de una superclase, es un error para el método de la subclase enlistar más excepciones en su cláusula `throws` que las del método sobrescrito de la superclase. Sin embargo, la cláusula `throws` de una subclase puede contener un subconjunto de la lista `throws` de una superclase.



Observación de ingeniería de software 15.7

Si su método llama a otros métodos que lancen explícitamente excepciones verificadas, esas excepciones deberán ser atrapadas o declaradas en su método. Si es posible, un método deberá atrapar una excepción, en vez de declararla.

El compilador de Java no verifica el código para determinar si una excepción no verificada es atrapada o declarada. Las excepciones tipo `Error` se consideran como no verificadas por el compilador, ya que generalmente no pueden ser manejadas por un programa.

Por lo general, las excepciones no verificadas pueden evitarse mediante una codificación apropiada. Por ejemplo, la expresión no verificada `ArithmeticException` (una subclase de `RuntimeException`) lanzada por el método `cociente` (líneas 76 a 80) en la figura 15.1 puede evitarse si el método asegura que el denominador no sea cero antes de intentar realizar la división. No se requiere que las excepciones no verificadas se enlisten en la cláusula `throws` de un método.

http://librosysolucionarios.net



Observación de ingeniería de software 15.8

Aunque el compilador no implementa el requerimiento de atrapar o declarar para las excepciones no verificadas, usted deberá proporcionar un código apropiado para el manejo de excepciones cuando sepa que dichas excepciones podrían ocurrir. Por ejemplo, un programa podría procesar excepciones `NumberFormatException` del método `parseInt` de la clase `Integer`, aun cuando las excepciones `NumberFormatException` sean no verificadas. (La clase `NumberFormatException` es una subclase de `RuntimeException`). Esto hará que sus programas sean más robustos.

El tipo de una excepción es el que determina si ésta es verificada o no. Todos los tipos de excepciones que son subclases directas o indirectas de la clase `RuntimeException` (paquete `java.lang`) son excepciones no verificadas.

De una superclase común pueden derivarse varias clases de excepciones. Si se escribe un manejador `catch` para atrapar objetos de excepción de un tipo de superclase, también se pueden atrapar todos los objetos de las subclases de esa clase. Esto permite que una cláusula `catch` maneje los errores relacionados con una notación concisa, y permite el procesamiento polimórfico de las excepciones relacionadas. Evidentemente, se podría atrapar a cada uno de los objetos de excepción de las subclases en forma individual, si estas excepciones requirieran un procesamiento distinto. Desde luego que atrapar excepciones relacionadas en una cláusula `catch` tendría sentido solamente si el comportamiento del manejo fuera el mismo para todas las subclases. De no ser así, se debe atrapar a cada una de las excepciones de las subclases en forma individual.



Tip para prevenir errores 15.3

Atrapar los objetos de excepción de las subclases en forma individual puede ocasionar errores si usted olvida evaluar uno o más de los tipos de subclase en forma explícita; al atrapar a la superclase se garantiza que se atraparán los objetos de todas las subclases. A menudo, una cláusula `catch` para el tipo de la superclase va después de las demás cláusulas `catch` para las subclases, para asegurar que todas las excepciones se procesen apropiadamente.

15.5 Volver a lanzar una excepción

Es posible que una cláusula `catch`, después de recibir una excepción, decida que no puede procesar esa excepción o que sólo puede procesarla parcialmente. En tales casos, la cláusula `catch` puede diferir el manejo de la excepción (o tal vez una porción de la misma) hacia otra cláusula `catch`. En cualquier caso, el manejador logra esto *volviendo a lanzar* la excepción, mediante la instrucción:

```
throw referenciaExcepcion;
```

en donde *referenciaExcepcion* es el nombre del parámetro para la excepción en el manejador `catch`.

Cuando se vuelve a lanzar la excepción, el siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, que el manejador `catch` que aparece después de ese bloque `try` circundante intenta manejar.



Buena práctica de programación 15.4

Si su método es capaz de manejar un tipo dado de excepción; entonces debe manejarla en vez de pasarla hacia otras regiones de su programa. Esto hace que los programas sean más claros.

15.6 Cláusula `finally`

Los programas que obtienen ciertos tipos de recursos deben devolver esos recursos al sistema en forma explícita, para evitar las denominadas *fugas de recursos*. En lenguajes de programación como C y C++, el tipo más común de fuga de recursos es la fuga de memoria. Java realiza la recolección automática de basura en la memoria que ya no es utilizada por los programas, evitando así la mayoría de las fugas de memoria. Sin embargo, pueden ocurrir otros tipos de fugas de recursos en Java. Por ejemplo, los archivos, las conexiones de bases de datos y conexiones de red que no se cierran apropiadamente podrían no estar disponibles para su uso en otros programas, o incluso más tarde en la ejecución del mismo programa.



Observación de ingeniería de software 15.9

Una cláusula `finally` generalmente contiene código para liberar los recursos adquiridos en su correspondiente bloque `try`; ésta es una manera efectiva de eliminar las fugas de recursos. Por ejemplo, la cláusula `finally` debe cerrar todos los archivos que se abran en el bloque `try`.

Tip para prevenir errores 15.4



Hay una pequeña cuestión en Java: no elimina completamente las fugas de memoria. Java no hace recolección de basura en un objeto, sino hasta que no existen más referencias a ese objeto. Por lo tanto, pueden ocurrir fugas de memoria, pero solamente si los programadores mantienen por error referencias a objetos no deseados. La mayoría de los problemas de fuga de memoria se resuelven mediante la recolección de basura de Java.

La cláusula `finally` es opcional. Si está presente, se coloca después de la última cláusula `catch`, como se muestra a continuación:

```
try {
    instrucciones
    instrucciones para adquirir recursos
}
catch ( UnTipoDeExcepción excepción1 ) {
    instrucciones para manejar excepciones
}
catch ( OtroTipoDeExcepción excepción2 ) {
    instrucciones para manejar excepciones
}
finally {
    instrucciones
    instrucciones para liberar recursos
}
```

Java garantiza que una cláusula `finally` (si hay una presente después de una secuencia `try/catch`) se ejecutará, se lance o no una excepción en el bloque `try` correspondiente, o en cualquiera de sus cláusulas `catch` correspondientes. Java también garantiza que una cláusula `finally` (si hay una presente) se ejecutará si un bloque `try` se sale mediante el uso de una instrucción `return`, `break` o `continue`.

El código para liberar recursos generalmente se coloca en una cláusula `finally`. Suponga que se asigna un recurso en un bloque `try`. Si no ocurre una excepción, se ignoran los manejadores `catch` y el control procede a la cláusula `finally`, la cual libera el recurso. El control del programa continúa con la primera instrucción que sigue después de la cláusula `finally`. Si ocurre una excepción, el programa ignora el resto del bloque `try`. Si el programa atrapa la excepción en uno de los manejadores `catch`, el programa procesa la excepción. Después, la cláusula `finally` libera el recurso y el control procede a la primera instrucción después de la cláusula `finally`.

Si una excepción que ocurre en el bloque `try` no puede ser atrapada por uno de los manejadores `catch` de ese bloque `try`, el programa ignora el resto del bloque `try` y el control procede a la cláusula `finally`, la cual libera el recurso. Después el programa pasa la excepción hacia arriba de la cadena de llamadas (es decir, hacia el método que hizo la llamada), en donde se trata de atraparla. Este proceso puede ocurrir muchas veces.

Si un manejador `catch` lanza una excepción, la cláusula `finally` se ejecuta de todas formas. Después la excepción se pasa al método que hizo la llamada.

La aplicación de Java de la figura 15.3 demuestra que la cláusula `finally` (si hay una presente después de una secuencia `try/catch`) se ejecutará, aun cuando no se lance una excepción en el bloque `try` correspondiente. El programa contiene los métodos `main` (líneas 5 a 17), `lanzarExcepcion` (líneas 20 a 44) y `noLanzaExcepcion` (líneas 47 a 67). Los métodos `lanzarExcepcion` y `noLanzaExcepcion` se declaran como `static`, de manera que `main` (otro método `static`) pueda llamarlos directamente.

El método `main` empieza a ejecutarse, entra a su bloque `try` e inmediatamente llama al método `lanzarExcepcion` (línea 8). El método `lanzarExcepcion` lanza una excepción `Exception` (línea 25), la atrapa (línea 29) y la vuelve a lanzar (línea 32). La excepción que se vuelve a lanzar se maneja en `main`, pero primero se ejecuta la cláusula `finally` (líneas 38 a 40). El método `main` detecta la excepción que vuelve a lanzarse en el bloque `try` en `main` (líneas 7 a 9) y la maneja mediante la cláusula `catch` (líneas 12 a 14). A continuación, `main` llama al método `noLanzaExcepcion` (línea 16). No se lanza ninguna excepción en el bloque `try` de `noLanzaExcepcion` (líneas 50 a 52), por lo que el programa ignora la cláusula `catch` (líneas 55 a 57), pero la cláusula `finally` (líneas 60 a 63) se ejecuta de todas maneras. El control procede a la instrucción que sigue después de la cláusula `finally`. Después el control regresa a `main` y el programa termina.

http://librosysolucionarios.net


```
1  // Fig. 15.3: UsoExcepciones.java
2  // Demostración del mecanismo try-catch-finally para manejar excepciones.
3  public class UsoExcepciones {
4
5     public static void main( String args[] )
6     {
7        try {
8           lanzarExcepcion(); // llamar al método lanzarExcepcion
9        }
10
11       // atrapar excepciones lanzadas por el método lanzarExcepcion
12       catch ( Exception excepcion ) {
13          System.err.println( "La excepcion se manejo en main" );
14       }
15
16       noLanzaExcepcion();
17    }
18
19    // demostrar try/catch/finally
20    public static void lanzarExcepcion() throws Exception
21    {
22       // lanzar una excepción y atraparla inmediatamente
23       try {
24          System.out.println( "El metodo lanzarExcepcion" );
25          throw new Exception();  // generar excepción
26       }
27
28       // atrapar la excepción lanzada en el bloque try
29       catch ( Exception excepcion ) {
30          System.err.println(
31             "La excepcion se manejo en el metodo lanzarExcepcion" );
32          throw excepcion;  // volver a lanzar para procesarla posteriormente
33
34          // cualquier código aquí no llegaría a ejecutarse
35       }
36
37       // este bloque se ejecuta, sin importar lo que ocurra en try/catch
38       finally {
39          System.err.println( "Finalmente se ejecuto en lanzarExcepcion" );
40       }
41
42       // cualquier código aquí no llegaría a ejecutarse
43
44    } // fin del método lanzarExcepcion
45
46    // demostrar finally cuando no ocurre excepción
47    public static void noLanzaExcepcion()
48    {
49       // el bloque try no lanza una excepción
50       try {
51          System.out.println( "El metodo noLanzaExcepcion" );
52       }
53
54       // catch no se ejecuta, porque no se lanzó una excepción
55       catch( Exception excepcion ) {
```

Figura 15.3 Mecanismo `try-catch-finally` para manejar excepciones. (Parte 1 de 2.)

```
56            System.err.println( excepcion );
57         }
58
59         // la cláusula this se ejecuta, sin importar lo que ocurra en try/catch
60         finally {
61            System.err.println(
62               "Finalmente se ejecuto en noLanzaExcepcion" );
63         }
64
65         System.out.println( "Fin del metodo noLanzaExcepcion" );
66
67      } // fin del método noLanzaExcepcion
68
69   } // fin de la clase UsoExcepciones
```

```
El metodo lanzarExcepcion
La excepcion se manejo en el metodo lanzarExcepcion
Finalmente se ejecuto en lanzarExcepcion
La excepcion se manejo en main
El metodo noLanzaExcepcion
Finalmente se ejecuto en noLanzaExcepcion
Fin del metodo noLanzaExcepcion
```

Figura 15.3 Mecanismo **try-catch-finally** para manejar excepciones. (Parte 2 de 2.)



Error común de programación 15.9

*Si una excepción no se ha atrapado cuando el control entra a una cláusula **finally**, y esta cláusula lanza una excepción que no se atrape en la cláusula **finally**, la primera excepción se perderá y la excepción de la cláusula **finally** será devuelta al método que hizo la llamada.*



Error común de programación 15.10

*Si suponemos que una excepción que se lance desde un manejador **catch** será procesada por ese manejador o cualquier otro manejador asociado con el mismo bloque **try**, se pueden ocasionar errores lógicos.*



Tip para prevenir errores 15.5

*Evite colocar código que pueda lanzar (**throw**) una excepción en una cláusula **finally**. Si se requiere dicho código, encierre el código en un bloque **try/catch** dentro de la cláusula **finally**.*



Buena práctica de programación 15.5

*El mecanismo de manejo de excepciones de Java está diseñado para eliminar el código de procesamiento de errores de la línea principal del código de un programa, para mejorar su claridad. No coloque bloques **try/catch/finally** alrededor de todas las instrucciones que puedan lanzar una excepción. Esto hace que los programas sean difíciles de leer. En vez de ello, ponga un bloque **try** alrededor de una porción considerable de su código, después de ese bloque **try** coloque cláusulas **catch** que manejen cada excepción posible, y después de las cláusulas **catch** escriba una sola cláusula **finally** (en caso de requerirse).*



Tip de rendimiento 15.2

Una regla general es que una vez que ya no sean necesarios los recursos deben liberarse lo más pronto posible; para que puedan reutilizarse de manera inmediata, lo cual redunda en un mejor rendimiento del programa.

15.7 Limpieza de la pila

Cuando se lanza una excepción, pero no se atrapa en un alcance específico, la pila de llamadas a métodos se limpia y se hace un intento de atrapar (**catch**) la excepción en la siguiente instrucción **try/catch** exterior.

http://librosysolucionarios.net

Limpiar la pila de llamadas a métodos significa que el método en el que no se atrapó la excepción termina, todas las variables en ese método quedan fuera de alcance y el control regresa a la instrucción que invocó originalmente a ese método. Si un bloque **try** encierra a esa instrucción, se hace un intento de atrapar (**catch**) esa excepción. Si un bloque **try** no encierra a esa instrucción, se lleva a cabo la limpieza de la pila otra vez. Si ningún manejador **catch** atrapa a esta excepción, todo el programa (o una parte del mismo) terminará,[3] El programa de la figura 15.4 demuestra la limpieza de la pila.

Cuando se ejecuta el método **main**, la línea 9 en el bloque **try** llama al método **lanzarExcepcion** (líneas 19 a 38). En el bloque **try** del método **lanzarExcepcion**, la línea 24 lanza una excepción **Exception**. Esto termina el bloque **try** inmediatamente, y el control pasa al manejador **catch** de la línea 28. El

```
1  // Fig. 15.4: UsoExcepciones.java
2  // Demostración de la limpieza de la pila.
3  public class UsoExcepciones {
4
5     public static void main( String args[] )
6     {
7        // llamar a lanzaExcepcion para demostrar la limpieza de la pila
8        try {
9           lanzaExcepcion();
10       }
11
12       // atrapar excepción lanzada en lanzaExcepcion
13       catch ( Exception excepcion ) {
14          System.err.println( "La excepcion se manejo en main" );
15       }
16    }
17
18    // lanzaExcepcion lanza la excepción que no es atrapada en este método
19    public static void lanzaExcepcion() throws Exception
20    {
21       // lanzar una excepción y atraparla en main
22       try {
23          System.out.println( "El metodo lanzaExcepcion" );
24          throw new Exception();       // generar excepción
25       }
26
27       // catch es del tipo incorrecto, por lo que la excepción no es atrapada
28       catch( RuntimeException excepcionTiempoEjecucion ) {
29          System.err.println(
30             "La excepcion se manejo en el metodo lanzaExcepcion" );
31       }
32
33       // la cláusula finally siempre se ejecuta
34       finally {
35          System.err.println( "Finally siempre se ejecuta" );
36       }
37
38    } // fin del método lanzaExcepcion
39
40 } // fin de la clase UsoExcepciones
```

Figura 15.4 Limpieza de la pila. (Parte 1 de 2.)

3. En el capítulo 16 hablaremos sobre el subprocesamiento múltiple, en el cual varias partes de un programa (subprocesos) se pueden ejecutar en paralelo. Como veremos, una excepción sin manejar puede terminar un subproceso de ejecución. Si ése es el único subproceso restante, el programa terminará.

```
El metodo lanzaExcepcion
Finally siempre se ejecuta
La excepcion se manejo en main
```

Figura 15.4 Limpieza de la pila. (Parte 2 de 2.)

tipo que va a atraparse (`RuntimeException`) no concuerda exactamente con el tipo lanzado (`Excepcion`) y no es una superclase del tipo lanzado, por lo que la excepción no se atrapa en el método `lanzarExcepcion`. La excepción debe manejarse para que el programa pueda continuar ejecutándose normalmente. Por lo tanto, el método `lanzarExcepcion` termina (pero sólo hasta que se ejecute su cláusula `finally`) y devuelve el control a la línea 9; el punto desde el que fue llamado en el programa. La línea 9 se encuentra en el bloque `try` circundante. La excepción aún no se ha manejado, por lo que el bloque `try` termina y se trata de atrapar la excepción en la línea 13. El tipo que va a atraparse (`Exception`) concuerda con el tipo lanzado. Por consecuencia, el manejador `catch` procesa la excepción y el programa termina al final de `main`.

15.8 `printStackTrace`, `getStackTrace` y `getMessage`

Recuerde que en la sección 15.4 vimos que las excepciones se derivan de la clase `Throwable`. Esta clase ofrece un método llamado ***`printStackTrace`***, el cual imprime la pila de llamadas a métodos. A menudo esto ayuda en la prueba y la depuración. La clase `Throwable` también proporciona un método llamado ***`getStackTrace`***, el cual obtiene la información que imprime `printStackTrace`. El método ***`getMessage`*** de la clase `Throwable` devuelve la cadena descriptiva almacenada en una excepción. En esta sección, consideraremos un ejemplo para demostrar estos tres métodos.

Tip para prevenir errores 15.6

Una excepción que no sea atrapada, de un momento a otro hará que se ejecute el manejador de excepciones predeterminado de Java. Éste muestra el nombre de la excepción, un mensaje y un rastreo completo de la pila de ejecución. La pila de ejecución muestra la pila de llamada a métodos (es decir, la cadena de llamadas) al momento en que ocurrió la excepción, dejando que el programador vea la ruta de ejecución que produjo la excepción archivo por archivo (y por lo tanto, clase por clase) y método por método. Esta información es útil para depurar un programa.

Tip para prevenir errores 15.7

El método `toString` de `Throwable` (heredado en todas las subclases de `Throwable`) devuelve una cadena que contiene el nombre de la clase y el mensaje descriptivo que indica el problema que ocurrió.

Buena práctica de programación 15.6

Nunca ignore una excepción que atrape. Cuando menos utilice a `printStackTrace` para mostrar un mensaje de error.

En la figura 15.5 se demuestra el uso de `getMessage`, `printStackTrace` y `getStackTrace`. El método `getMessage` devuelve la cadena descriptiva almacenada en una excepción. El método `printStackTrace` muestra en el flujo de error estándar (que generalmente es la línea de comandos o la consola) un mensaje de error con el nombre de la clase de la excepción, la cadena descriptiva almacenada en la excepción y una lista de los métodos que no terminaron de ejecutarse cuando se lanzó la excepción (es decir, todos los métodos que residen actualmente en la pila de llamadas a métodos). Si queremos mostrar la información de rastreo de la pila a flujos que no sean el flujo de error estándar, podemos utilizar `getStackTrace`.

En `main`, el bloque `try` (líneas 7 a 9) llama a `metodo1` (declarado en las líneas 37 a 40). A continuación, `metodo1` llama a `metodo2` (declarado en las líneas 43 a 46), el cual a su vez llama a `metodo3` (declarado en las líneas 49 a 52). En la línea 51 de `metodo3` se lanza un objeto `Exception`. Sin embargo, y como ningún bloque `try` encierra a la instrucción `throw` de la línea 51, se lleva a cabo la limpieza de la pila; `metodo3` termina en la línea 51 y después regresa el control a la instrucción en `metodo2` que invocó a `metodo3` (es decir, la línea 45). Como ningún bloque `try` encierra a la línea 45, se lleva a cabo la limpieza de la


```
1  // Fig. 15.5: UsoExcepciones.java
2  // Demostración de getMessage y printStackTrace de la clase Exception.
3  public class UsoExcepciones {
4
5     public static void main( String args[] )
6     {
7        try {
8           metodo1(); // llamar a metodo1
9        }
10
11       // atrapar las excepciones lanzadas desde metodo1
12       catch ( Exception excepcion ) {
13          System.err.println( excepcion.getMessage() + "\n" );
14          excepcion.printStackTrace();
15
16          // obtener la información de rastreo de la pila
17          StackTraceElement[] elementosRastreo = excepcion.getStackTrace();
18
19          System.out.println(  "\nRastreo de pila proveniente de getStackTrace:" );
20          System.out.println( "Clase\t\tArchivo\t\t\tLinea\tMetodo" );
21
22          // iterar a través de elementosRastreo para obtener descripción de las excepciones
23          for ( int i = 0; i < elementosRastreo.length; i++ ) {
24           StackTraceElement elementoActual = elementosRastreo[ i ];
25             System.out.print( elementoActual.getClassName() + "\t" );
26             System.out.print( elementoActual.getFileName() + "\t" );
27             System.out.print( elementoActual.getLineNumber() + "\t" );
28             System.out.print( elementoActual.getMethodName() + "\n" );
29
30          } // fin de la instrucción for
31
32       } // fin de la instrucción catch
33
34    } // fin del método main
35
36    // llamar a metodo2; lanzar excepciones de vuelta hacia main
37    public static void metodo1() throws Exception
38    {
39       metodo2();
40    }
41
42    // llamar al método3; lanzar excepciones de vuelta hacia metodo1
43    public static void metodo2() throws Exception
44    {
45       metodo3();
46    }
47
48    // lanzar excepción de vuelta a metodo2
49    public static void metodo3() throws Exception
50    {
51       throw new Exception( "La excepcion se lanzo en metodo3" );
52    }
53
54 } // fin de la clase UsoExcepciones
```

Figura 15.5 Los métodos `getMessage`, `getStackTrace` y `printStackTrace` de `Throwable`. (Parte 1 de 2.)

http://librosysoluciones.net

```
La excepcion se lanzo en metodo3

java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoExcepciones.metodo3(UsoExcepciones.java:51)
        at UsoExcepciones.metodo2(UsoExcepciones.java:45)
        at UsoExcepciones.metodo1(UsoExcepciones.java:39)
        at UsoExcepciones.main(UsoExcepciones.java:8)

Rastreo de pila proveniente de getStackTrace:
Clase           Archivo                 Linea   Metodo
UsoExcepciones  UsoExcepciones.java     51      metodo3
UsoExcepciones  UsoExcepciones.java     45      metodo2
UsoExcepciones  UsoExcepciones.java     39      metodo1
UsoExcepciones  UsoExcepciones.java     8       main
```

Figura 15.5 Los métodos `getMessage`, `getStackTrace` y `printStackTrace` de `Throwable`. (Parte 2 de 2.)

pila otra vez; `metodo2` termina en la línea 45 y regresa el control a la instrucción en `metodo1` que invocó a `metodo2` (es decir, la línea 39). Como ningún bloque `try` encierra a la línea 39, se lleva a cabo la limpieza de la pila una vez más; `metodo1` termina en la línea 39 y regresa el control a la instrucción en `main` que invocó a `metodo1` (es decir, la línea 8). El bloque `try` de las líneas 7 a 9 encierra a esta instrucción. La excepción no ha sido manejada, por lo que el bloque `try` termina y el primer manejador `catch` (líneas 12 a 32) concordante atrapa y procesa la excepción.

En la línea 13 se invoca al método `getMessage` de la excepción, para obtener la descripción de la misma. En la línea 14 se invoca al método `printStackTrace` de la excepción, para mostrar el rastreo de la pila, el cual indica en dónde ocurrió la excepción. En la línea 17 se invoca al método `printStackTrace` de la excepción, para mostrar el rastreo de la pila, el cual indica en dónde ocurrió la excepción como un arreglo de objetos `StackTraceElement`. En las líneas 23 a 30 se obtiene cada uno de los objetos `StackTraceElement` en el arreglo, y se invocan sus métodos `getClassName`, `getFileName`, `getLineNumber` y `getMethodName` para obtener el nombre de la clase, el nombre del archivo, el número de línea y el nombre del método, respectivamente, para ese objeto `StackTraceElement`. Cada objeto `StackTraceElement` representa la llamada a un método en la pila de llamadas a métodos.

Los resultados de la figura 15.5 muestran que la información de rastreo de la pila que imprime `printStackTrace` sigue el patrón: *nombreClase.nombreMétodo*(*nombreArchivo*: *númeroLínea*), en donde *nombreClase*, *nombreMétodo* y *nombreArchivo* indican los nombres de la clase, el método y el archivo en los que ocurrió la excepción, y *númeroLínea* indica en dónde ocurrió la excepción. El método `getStackTrace` permite un procesamiento personalizado de la información sobre la excepción. Compare la salida de `printStackTrace` con la salida creada a partir de los objetos `StackTraceElement`, y podrá ver que ambos contienen la misma información de rastreo de la pila.

15.9 Excepciones encadenadas

Es relativamente común para un manejador `catch` el atrapar un tipo de excepción y después lanzar una nueva excepción de un tipo distinto, para indicar una excepción específica del programa que haya ocurrido. En las primeras versiones de Java, no había mecanismo para envolver la información de la excepción original con la información de la nueva excepción, para proporcionar un rastreo completo de la pila, indicando en dónde ocurrió el problema original en el programa. Esto hacía que depurar dichos problemas fuera un proceso bastante difícil. La Plataforma Java 2, Standard Edition, versión 1.4 proporciona *excepciones encadenadas* para permitir que un objeto de excepción mantenga la información completa sobre el rastreo de la pila. En la figura 15.6 presentamos un ejemplo mecánico que demuestra el funcionamiento de las excepciones encadenadas.

El programa consiste de cuatro métodos: `main` (líneas 5 a 15), `metodo1` (líneas 18 a 28), `metodo2` (líneas 31 a 41) y `metodo3` (líneas 44 a 47). La línea 8 en el bloque `try` de `main` llama a `metodo1`. La línea

http://librosysolucionarios.net

21 en el bloque `try` de `metodo1` llama a `metodo2`. La línea 34 en el bloque `try` de `metodo2` llama a `metodo3`. En `metodo3`, la línea 46 lanza una nueva excepción `Exception`. Como esta instrucción no se encuentra dentro de un bloque `try`, el `metodo3` termina y la excepción se devuelve al método que hace la lla-

```
1  // Fig. 15.6: UsoExcepcionesEncadenadas.java
2  // Demostración de las excepciones encadenadas.
3  public class UsoExcepcionesEncadenadas {
4
5     public static void main( String args[] )
6     {
7        try {
8           metodo1(); // llamar a metodo1
9        }
10
11       // atrapar las excepciones lanzadas desde metodo1
12       catch ( Exception excepcion ) {
13          excepcion.printStackTrace();
14       }
15    }
16
17    // llamar al metodo2; lanzar excepciones de vuelta a main
18    public static void metodo1() throws Exception
19    {
20       try {
21          metodo2(); // llamar a metodo2
22       }
23
24       // atrapar la excepción lanzada desde metodo2
25       catch ( Exception excepcion ) {
26          throw new Exception( "Excepcion lanzada en metodo1", excepcion );
27       }
28    }
29
30    // llamar a metodo3; lanzar excepciones de vuelta a metodo1
31    public static void metodo2() throws Exception
32    {
33       try {
34          metodo3(); // llamar a metodo3
35       }
36
37       // atrapar la excepción lanzada desde metodo3
38       catch ( Exception excepcion ) {
39          throw new Exception( "Excepcion lanzada en metodo2", excepcion );
40       }
41    }
42
43    // lanzar excepción de vuelta a metodo2
44    public static void metodo3() throws Exception
45    {
46       throw new Exception( "Excepcion lanzada en metodo3" );
47    }
48
49 } // fin de la clase UsoExcepcionesEncadenadas
```

Figura 15.6 Excepciones encadenadas. (Parte 1 de 2.)

```
java.lang.Exception: Excepcion lanzada en metodo1
        at UsoExcepcionesEncadenadas.metodo1(UsoExcepcionesEncadenadas.java:26)
        at UsoExcepcionesEncadenadas.main(UsoExcepcionesEncadenadas.java:8)
Caused by: java.lang.Exception: Excepcion lanzada en metodo2
        at UsoExcepcionesEncadenadas.metodo2(UsoExcepcionesEncadenadas.java:39)
        at UsoExcepcionesEncadenadas.metodo1(UsoExcepcionesEncadenadas.java:21)
        ... 1 more
Caused by: java.lang.Exception: Excepcion lanzada en metodo3
        at UsoExcepcionesEncadenadas.metodo3(UsoExcepcionesEncadenadas.java:46)
        at UsoExcepcionesEncadenadas.metodo2(UsoExcepcionesEncadenadas.java:34)
        ... 2 more
```

Figura 15.6 Excepciones encadenadas. (Parte 2 de 2.)

mada (`metodo2`), en la línea 34. Esta instrucción se encuentra dentro de un bloque `try`; por lo tanto, el bloque `try` termina y la excepción es atrapada en las líneas 38 a 40. En la línea 39, en el manejador `catch`, se lanza una nueva excepción. En este caso, se hace una llamada al constructor `Exception` con dos argumentos (el cual es nuevo en J2SE 1.4). El segundo argumento representa a la excepción que era la causa original del problema que ocurrió. En este programa, esa excepción ocurrió en la línea 45. Como se lanza una excepción desde el manejador `catch`, el `metodo2` termina y devuelve la nueva excepción al método que hace la llamada (`metodo1`), en la línea 21. Una vez más, esta instrucción se encuentra dentro de un bloque `try`, por lo tanto, este bloque termina y la excepción es atrapada en las líneas 25 a 27. En la línea 26, en el manejador `catch` se lanza una nueva excepción y se utiliza la excepción que se atrapó como el segundo argumento para el constructor de `Exception`. Como se lanza una excepción desde el manejador `catch`, el `metodo1` termina y devuelve la nueva excepción al método que hace la llamada (`main`), en la línea 8. El bloque `try` en `main` termina y la excepción es atrapada en las líneas 12 a 14. En la línea 13 se imprime el rastreo de la pila.

Observe en la salida del programa que las primeras tres líneas muestran la excepción más reciente que fue lanzada (es decir, la del `metodo1` en la línea 26). Las siguientes cuatro líneas indican la excepción que se lanzó desde el `metodo2`, en la línea 39. Por último, las siguientes cuatro líneas representan la excepción que se lanzó desde el `metodo3`, en la línea 46. Además observe que, si lee la salida en forma inversa, muestra cuántas excepciones encadenadas más quedan pendientes.

15.10 Declaración de nuevos tipos de excepciones

La mayoría de los programadores de Java utilizan las clases existentes de la API de Java (o de distribuidores independientes) para crear aplicaciones de Java. Los métodos de esas clases por lo general se declaran para lanzar las excepciones apropiadas cuando ocurren problemas. Los programadores escriben código para procesar esas excepciones existentes, para que sus programas sean más robustos.

Si usted es un programador y crea clases que otros programadores utilizarán en sus programas, tal vez le sea conveniente declarar sus propias clases de excepciones que sean específicas para los problemas que pueden ocurrir cuando otro programador utilice sus clases reutilizables.



Observación de ingeniería de software 15.10

De ser posible, indique las excepciones de sus métodos mediante el uso de un tipo de excepción existente, en vez de crear una nueva clase. La API de Java contiene muchos tipos de excepciones que podrían ser adecuados para el tipo de problema que su método necesite indicar.

Una nueva clase de excepción debe extender a una clase de excepción existente, para poder asegurar que la clase pueda utilizarse con el mecanismo de manejo de excepciones. Al igual que cualquier otra clase, una clase de excepción puede contener campos y métodos. Sin embargo, una nueva clase de excepción por lo general contiene sólo dos constructores; uno que no toma argumentos y pasa un mensaje de excepción predeterminado al constructor de la superclase, y otro que recibe un mensaje de excepción personalizado y lo pasa al constructor de la superclase.

http://librosysolucionarios.net



Buena práctica de programación 15.7

Asociar cada uno de los tipos de fallas graves en tiempo de ejecución con una clase de excepción con nombre ayuda a mejorar la claridad del programa.



Observación de ingeniería de software 15.11

Al definir su propio tipo de excepción, estudie las clases de excepción existentes en la API de Java y extienda una clase de excepción relacionada. Por ejemplo, si va a crear una nueva clase para representar cuando un método intenta realizar una división entre cero, podría extender la clase `ArithmeticException`, ya que la división entre cero ocurre durante la aritmética. Si las clases existentes no son superclases apropiadas para su nueva clase de excepción, debe decidir si su nueva clase debe ser una clase de excepción verificada o no verificada. La nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a `Exception`) si el programa debe manejar la excepción. El programa debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. La nueva clase debe extender a `RuntimeExcepcion` si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada).

En el capítulo 20 proporcionaremos un ejemplo de una clase de excepción personalizada. Declararemos una clase reutilizable llamada `Lista`, la cual es capaz de almacenar una lista de referencias a objetos. Algunas operaciones que se realizan comúnmente en una `Lista` no se permitirán si la `Lista` está vacía. Por esta razón, algunos métodos de `Lista` lanzan excepciones de la clase de excepción `ListaVaciaException`.



Buena práctica de programación 15.8

Por convención, todos los nombres de las clases de excepción deben terminar con la palabra `Exception`.

15.11 Constructores y manejo de excepciones

¿Qué ocurre cuando se detecta un error en un constructor? Por ejemplo, ¿cómo debe responder el constructor de un objeto cuando `new` falla e indica que no pudo asignar la memoria requerida para guardar la representación interna de ese objeto? Como el constructor no puede devolver un valor para indicar un error, debemos seleccionar un medio alternativo para indicar que el objeto no se creó apropiadamente. Un esquema es devolver el objeto creado incorrectamente y esperar que cualquiera que use ese objeto haga las pruebas apropiadas para determinar que el objeto exhibe un estado inconsistente. Otro esquema es establecer cierta variable fuera del constructor, que indique si el objeto está listo para usarse. Tal vez la mejor alternativa sea requerir que el constructor lance (mediante `throw`) una excepción, indicando el problema exacto que ocurrió, ofreciendo así una oportunidad para manejar la falla.

En este capítulo demostramos los conceptos y construcciones básicas del manejo de excepciones. Los capítulos restantes en el libro demuestran características del lenguaje y APIs más avanzadas. Como veremos, muchas de estas características del lenguaje y APIs generan excepciones verificadas cuando ocurren los problemas. Las técnicas que usted aprendió aquí serán importantes para crear programas robustos de Java, que en muchos casos puedan recuperarse de los problemas que ocurren durante la ejecución de un programa.

RESUMEN

- Algunos ejemplos comunes de excepciones son un índice de arreglo fuera de rango, el desbordamiento aritmético, la división entre cero y los parámetros inválidos de método, así como determinar que no hay memoria suficiente para satisfacer una solicitud de asignación mediante el operador `new`.
- El espíritu detrás del manejo de excepciones es permitir a los programas atrapar y manejar los errores, en vez de dejar que ocurran y sufrir las consecuencias.
- El manejo de excepciones está diseñado para tratar con errores sincrónicos (es decir, errores que ocurran como resultado de la ejecución de un programa).
- El manejo de excepciones no está diseñado para tratar con situaciones asíncronas, como llegadas de mensajes de red, completar las operaciones de E/S de disco, clics del ratón y demás.
- El manejo de excepciones generalmente se utiliza en situaciones en las que el problema será tratado por una parte distinta (es decir, un alcance distinto) a la parte del programa que detectó el problema y lanzó la excepción.

- Las excepciones no deben utilizarse como un mecanismo para especificar el flujo del control. Por lo general, el flujo de control con estructuras de control convencionales es más claro y eficiente que con las excepciones.
- El manejo de excepciones debe utilizarse para procesar las excepciones de los métodos, en donde no tenga sentido que esos métodos manejen sus propias excepciones.
- El manejo de excepciones debe utilizarse en proyectos grandes para manejar el procesamiento de errores de una manera uniforme para todo el proyecto.
- El manejo de excepciones en Java está dirigido a situaciones en las que el método que detecta un problema es incapaz de manejarlo. Dicho método lanzará (`throw`) una excepción. Si la excepción concuerda con el tipo del parámetro en una de las cláusulas `catch`, se ejecuta el código para esa cláusula `catch`.
- El programador encierra en un bloque `try` el código que puede generar una excepción, junto con el código que no debe ejecutarse en caso de que ocurra una excepción. El bloque `try` va seguido por una o más cláusulas `catch`. Cada cláusula `catch` es un manejador de excepciones que especifica el tipo de excepción que puede manejar.
- Si no se lanzan excepciones en un bloque `try`, se ignoran las cláusulas `catch` para ese bloque. Luego, el programa reanuda la ejecución después de la última cláusula `catch`, o después de ejecutar una cláusula `finally` (en caso de que se proporcione una).
- El operando de una instrucción `throw` puede ser de cualquier clase derivada de `Throwable`. Las subclases inmediatas de `Throwable` son `Error` y `Exception`.
- La clase `Exception` y sus subclases representan situaciones excepcionales en un programa de Java, y pueden ser atrapadas por el programa.
- La clase `Error` y sus subclases representan situaciones excepcionales en el sistema en tiempo de ejecución en Java, que generalmente no deben ser atrapadas por un programa.
- Se dice que las excepciones `RuntimeException` y `Error` son "no verificadas". Se dice que todos los demás tipos de excepciones son "verificadas". Las excepciones verificadas que son lanzadas por cierto método deben especificarse en la cláusula `throws` de ese método.
- Las excepciones son atrapadas por la cláusula `catch` (para el bloque `try` desde el que se lanzó la excepción), especificando un tipo apropiado.
- Una excepción termina el bloque en el que ocurrió la excepción.
- `catch( Exception excepcion )` atrapa a todas las excepciones `Exception`. `catch(Throwable lanzable)` atrapa a todas las excepciones `Exception` y `Error`.
- Un manejador puede volver a lanzar el objeto hacia un bloque `try` externo. Además, si no hay un manejador que concuerde con un objeto específico que se haya lanzado, la búsqueda de una concordancia continúa en un bloque `try` circundante.
- El objeto de una subclase puede ser atrapado, ya sea mediante un manejador que especifique a ese tipo de subclase, o mediante manejadores que especifiquen los tipos de cualquiera de sus superclases.
- Una cláusula `catch` no puede utilizar las variables que estén en el alcance de su bloque `try` ya que, para cuando la cláusula `catch` empieza a ejecutarse, el bloque `try` ha terminado. La información que necesita el manejador para procesar la excepción, generalmente se pasa en el objeto lanzado.
- Los errores de compilación se producen si las cláusulas `catch` para los tipos de la superclase aparecen antes que las cláusulas `catch` para los tipos de la subclase.
- Cuando se atrapa una excepción, es posible que puedan haberse asignado los recursos, pero que no se hayan liberado todavía en el bloque `try`. Una cláusula `finally` debe liberar a esos recursos.
- Es posible que el manejador que atrapa una excepción decida que no puede procesar esa excepción. En este caso, el manejador puede simplemente volver a lanzar la excepción. Una instrucción `throw`, seguida por el nombre del parámetro de excepción, vuelve a lanzar la excepción.
- Incluso si un manejador puede procesar una excepción, y sin importar que realice algún procesamiento en esa excepción, el manejador puede volver a lanzar la excepción para procesarla posteriormente, fuera del manejador. Una excepción que se volvió a lanzar se detecta mediante el siguiente bloque `try` circundante (normalmente en un método que hace la llamada) y se maneja mediante una cláusula `catch` apropiada (si hay una) para ese bloque `try` circundante.
- Una cláusula `throws` enlista las excepciones verificadas que pueden lanzarse desde un método. Un método puede lanzar (mediante `throw`) las excepciones indicadas, o puede lanzar tipos de subclases. Si se lanza una excepción verificada que no esté enlistada en la cláusula `throws`, se produce un error de compilación.

- Las excepciones encadenadas permiten que un nuevo objeto de excepción envuelva a un objeto de excepción existente, como parte de la nueva excepción. El rastreo de la pila contiene información acerca de en dónde ocurrió la excepción original, y en dónde fue lanzada la nueva excepción.

TERMINOLOGÍA

agotamiento de memoria
`ArithmeticException`
`ArrayIndexOutOfBoundsException`
atrapar todas las excepciones
atrapar una excepción
`catch ( Exception e )`
`catch`, cláusula
computación crítica para los negocios
computación de misión crítica
declarar las excepciones que pueden lanzarse
errores sincrónicos
`Error`, clase
excepción
excepción encadenada
excepción `Exception` no verificada
excepción verificada
excepción como un arreglo
`Exception`, clase
`finally`, cláusula
fuga de recursos
`getMessage`, método de `Throwable`
`getStackTrace`, método de `Throwable`
`IOException`
jerarquía de herencia para la clase `Throwable`
lanzar una excepción
limpieza de la pila
manejador de excepciones
manejador de excepciones predeterminado
manejar una excepción
manejo de errores
manejo de excepciones
modelo de reanudación del manejo de excepciones
modelo de terminación del manejo de excepciones
`null`, referencia
objeto de excepción
`printStackTrace`, método de `Throwable`
punto de lanzamiento
requerimiento de atrapar o declarar
`RuntimeException`
`StackTraceElement`
`throw`, instrucción
`Throwable`, clase
`throws`, cláusula
tolerancia a fallas
`try`, bloque
volver a lanzar una excepción

EJERCICIOS DE AUTOEVALUACIÓN

15.1 Enliste cinco ejemplos comunes de excepciones.

15.2 Dé varias razones por las cuales no deban utilizarse las técnicas de manejo de excepciones para el control convencional de los programas.

15.3 ¿Por qué son las excepciones particularmente apropiadas para tratar con los errores producidos por los métodos de las clases en la API de Java?

15.4 ¿Qué es una "fuga de recursos"?

15.5 Si no se lanzan excepciones en un bloque `try`, ¿hacia dónde procede el control cuando el bloque `try` completa su ejecución?

15.6 Mencione una ventaja clave del uso de `catch( Exception e )`.

15.7 ¿Debe un subprograma (applet) o aplicación convencional atrapar los objetos `Error`?

15.8 ¿Qué ocurre si ningún manejador `catch` concuerda con el tipo de un objeto lanzado?

15.9 ¿Qué ocurre si varios manejadores concuerdan con el tipo del objeto lanzado?

15.10 ¿Por qué debería un programador especificar un tipo de subclase como el tipo en un manejador `catch`?

15.11 ¿Cuál es la principal razón de utilizar cláusulas `finally`?

15.12 ¿Qué ocurre cuando un manejador `catch` lanza una excepción `Exception`?

15.13 ¿Qué hace la instrucción `throw` *referenciaExcepcion*?

15.14 ¿Qué ocurre a una referencia local en un bloque `try`, cuando ese bloque lanza una excepción `Exception`?

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

15.1 Agotamiento de memoria, índice de arreglo fuera de límites, desbordamiento aritmético, división entre cero, parámetros inválidos de método.

15.2 (a) El manejo de excepciones está diseñado para manejar las situaciones que ocurren con poca frecuencia y que a menudo provocan la terminación del programa, por lo que los escritores de compiladores no tienen que implementar el manejo de excepciones para funcionar de manera óptima. (b) Por lo general, el flujo de control con estructuras de control convencionales es más claro y eficiente que con las excepciones. (c) Pueden ocurrir problemas, ya

http://librosysolucionarios.net

que la pila se limpia cuando ocurre una excepción y los recursos asignados antes de la excepción tal vez no se liberen. (d) Las excepciones "adicionales" pueden interponerse en el camino de las excepciones de tipos de errores genuinos. Es más difícil para el programador llevar el registro de un número más extenso de casos de excepciones.

15.3 Es muy poco probable que los métodos de clases en la API de Java puedan realizar un procesamiento de errores que cumpla con las necesidades únicas de todos los usuarios.

15.4 Una "fuga de recursos" ocurre cuando un programa en ejecución no libera apropiadamente un recurso cuando éste ya no es necesario. Si el programa intenta utilizar el recurso de nuevo en el futuro, el programa tal vez no pueda acceder a ese recurso.

15.5 Las cláusulas `catch` para esa instrucción `try` se ignoran y el programa reanuda su ejecución después de la última cláusula `catch`. Si hay una cláusula `finally`, se ejecuta primero y luego el programa reanuda su ejecución después de la cláusula `finally`.

15.6 La forma `catch( Exception e )` atrapa cualquier tipo de excepción lanzada en una instrucción `try`. Una ventaja es que ninguna excepción `Exception` lanzada puede escabullirse sin que sea por lo menos atrapada. El programador puede entonces decidir si va a manejar la excepción o si posiblemente vuelva a lanzarla.

15.7 Las excepciones `Error` son generalmente problemas graves con el sistema de Java subyacente; en la mayoría de los programas no es conveniente atrapar excepciones `Error`, ya que el programa no podrá recuperarse de dichos problemas.

15.8 Esto hace que la búsqueda de una concordancia continúe en la siguiente instrucción `try` circundante.

15.9 Se ejecuta la primera cláusula `catch` concordante después del bloque `try`.

15.10 Esto permite a un programa atrapar tipos relacionados de excepciones, y procesarlos en una manera uniforme. Sin embargo, a menudo es conveniente procesar los tipos de subclases en forma individual, para un manejo de excepciones más preciso.

15.11 La cláusula `finally` es el medio preferido para evitar las fugas de recursos.

15.12 La excepción será procesada por un manejador `catch` (si existe uno) asociado con un bloque `try` circundante (si existe uno).

15.13 Vuelve a lanzar la excepción para procesar un manejador de excepciones de un bloque `try` circundante.

15.14 La referencia queda fuera de alcance, y la cuenta de referencias para el objeto se decrementa. Si la cuenta de referencias se vuelve cero, el objeto se marca para la recolección de basura.

EJERCICIOS

15.15 Enliste las diversas condiciones excepcionales que han ocurrido en programas, a lo largo de este texto. Enliste todas las condiciones excepcionales adicionales que pueda. Para cada una de ellas, describa brevemente la manera en que un programa manejaría la excepción, utilizando las técnicas de manejo de excepciones que se describen en este capítulo. Algunas excepciones típicas son la división entre cero, el desbordamiento aritmético, el índice de arreglo fuera de límites, etcétera.

15.16 Hasta este capítulo, hemos visto que tratar con los errores detectados por los constructores es algo difícil. Explique por qué el manejo de excepciones es un medio efectivo para tratar con las fallas en los constructores.

15.17 Suponga que un programa lanza una excepción, y que el manejador de excepciones apropiado empieza a ejecutarse. Ahora suponga que el mismo manejador de excepciones lanza la misma excepción. ¿Crea esto una recursividad infinita? Explique su respuesta.

15.18 Use la herencia para crear una superclase de excepción y varias subclases de excepción. Escriba un programa para demostrar que la instrucción `catch`, que especifica la superclase, atrape a las excepciones de las subclases.

15.19 Escriba un programa que demuestre cómo se atrapan las diversas excepciones mediante

```
catch ( Exception excepcion )
```

15.20 Escriba un programa que demuestre que el orden de las cláusulas `catch` es importante. Si trata de atrapar un tipo de excepción de superclase antes de un tipo de subclase, el compilador debe generar errores. Explique por qué ocurren estos errores.

15.21 Escriba un programa que demuestre cómo un constructor pasa información sobre la falla del constructor a un manejador de excepciones.

15.22 Escriba un programa que muestre cómo volver a lanzar una excepción.

15.23 Escriba un programa que demuestre que un método con su propio bloque `try` no tiene que atrapar cada posible error generado dentro de este bloque `try`. Algunas excepciones podrían escabullirse y manejarse en otros alcances.

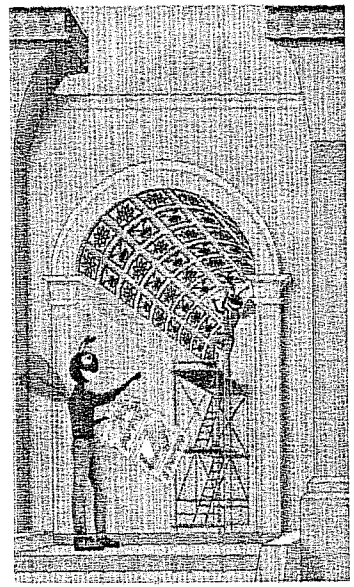
http://librosysolucionarios.net

16

Subprocesamiento múltiple

Objetivos

- Comprender la programación con subprocesamiento múltiple.
- Apreciar cómo el subprocesamiento puede mejorar el rendimiento de un programa.
- Comprender el ciclo de vida de un subproceso.
- Comprender las prioridades y la programación de subprocesos.
- Crear, administrar y destruir subprocesos.
- Comprender la sincronización de subprocesos.
- Comprender los subprocesos tipo "daemon".
- Detener y suspender subprocesos.
- Mostrar la salida de varios subprocesos en una GUI de Swing.



El toque de la araña, ¡tan exquisitamente fino!
Se siente en cada hilo y vive a lo largo de la línea.
Alexander Pope

Una persona con un reloj sabe qué hora es; una persona con dos relojes nunca estará segura.
Proverbio

Aprenda a laborar y esperar.
Henry Wadsworth Longfellow

La definición más general de la belleza...múltiple deidad en la unidad.
Samuel Taylor Coleridge

Plan general

16.1 Introducción
16.2 Estados de los subprocesos: Ciclo de vida de un subproceso
16.3 Prioridades y programación de subprocesos
16.4 Creación y ejecución de subprocesos
16.5 Sincronización de subprocesos
16.6 Relación productor/consumidor sin sincronización
16.7 Relación productor/consumidor con sincronización
16.8 Relación productor/consumidor: Búfer circular
16.9 Subprocesos tipo "daemon"
16.10 Interfaz `Runnable`
16.11 (Ejemplo práctico opcional) Acerca de los objetos: Subprocesamiento múltiple
16.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño de concurrencia

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

16.1 Introducción

Sería bueno si pudiéramos realizar una acción a la vez, y realizarla bien, pero por lo general eso es difícil. El cuerpo humano realiza una gran variedad de operaciones *en paralelo*; o, como diremos en este capítulo, *operaciones concurrentes*. Por ejemplo, la respiración, la circulación de la sangre y la digestión pueden ocurrir de manera concurrente. Todos los sentidos (vista, tacto, olfato, gusto y oído) pueden emplearse al mismo tiempo. Las computadoras también pueden realizar operaciones concurrentes.[1] Para las computadoras personales de escritorio es tarea común compilar un programa, enviar un archivo a una impresora y recibir mensajes de correo electrónico a través de una red, de manera concurrente.

La mayoría de los lenguajes de programación no permiten a los programadores especificar actividades concurrentes. Generalmente, los lenguajes de programación proporcionan sólo un simple conjunto de estructuras de control, las cuales permiten a los programadores realizar una acción a la vez, procediendo a la siguiente acción una vez que la anterior haya terminado. Históricamente, la concurrencia se ha implementado en forma de funciones primitivas del sistema operativo, disponibles sólo para los programadores experimentados.

El lenguaje de programación Ada, desarrollado por el Departamento de defensa de los Estados Unidos, hizo que las primitivas de concurrencia estuvieran disponibles ampliamente para los contratistas de defensa dedicados a la construcción de sistemas militares de comando y control. Sin embargo, Ada no ha sido utilizado de manera general en universidades o en la industria comercial.

Java pone las primitivas de concurrencia a disposición del programador de aplicaciones. El programador especifica que las aplicaciones contienen *subprocesos de ejecución*, en donde cada subproceso designa una porción de un programa que pueda ejecutarse concurrentemente con otros subprocesos. Esta capacidad, llamada *subprocesamiento múltiple*, ofrece al programador de Java poderosas herramientas que no están disponibles en C y C++, lenguajes en los cuales se basa Java. (*Nota:* En muchas plataformas computacionales, los programas de C y C++ pueden realizar el subprocesamiento múltiple mediante el uso de bibliotecas de código específicas para cada plataforma.)

1. Sólo las computadoras con múltiples procesadores pueden ejecutar operaciones concurrentes. Los sistemas operativos en computadoras con un solo procesador utilizan varias técnicas para simular la concurrencia, pero sólo pueden ejecutar un subproceso individual en un momento dado.

http://librosysolucionarios.net

Tip de rendimiento 16.1



Un problema con las aplicaciones de un solo subproceso es que las actividades que llevan mucho tiempo deben completarse antes de que puedan comenzar otras. En una aplicación con subprocesamiento múltiple, los subprocesos pueden distribuirse entre varios procesadores, de manera que se realicen varias tareas concurrentemente. El subprocesamiento múltiple también puede incrementar el rendimiento en sistemas con un solo procesador que simulan la concurrencia.

Observación de ingeniería de software 16.1



A diferencia de los lenguajes que no tienen capacidades integradas de subprocesamiento múltiple (como C y C++) y que por lo tanto deben realizar llamadas no portables a las primitivas de subprocesamiento múltiple del sistema operativo, Java incluye las primitivas de subprocesamiento múltiple como parte del mismo lenguaje y como parte de las clases del paquete `java.lang` para manipular subprocesos de una manera transparente entre plataformas.

Hablaremos sobre muchas aplicaciones de la programación concurrente. Por ejemplo, cuando los programas descargan archivos extensos como clips de audio o vídeo desde World Wide Web, lo usuarios no desean esperar hasta que se descargue todo un clip completo para empezar a reproducirlo. Para resolver este problema podemos poner varios subprocesos a trabajar; un subproceso descarga un clip y el otro lo reproduce. Estas actividades, o *tareas*, se llevan a cabo concurrentemente. Para evitar que la reproducción del clip tenga interrupciones, *sincronizamos* los subprocesos de manera que el subproceso de reproducción no empiece sino hasta que haya una cantidad suficiente del clip en memoria, como para mantener ocupado al subproceso de reproducción.

Otro ejemplo de subprocesamiento múltiple es la *recolección de basura*, automática en Java. C y C++ requieren que el programador reclame en forma explícita la memoria asignada dinámicamente. Java proporciona un *subproceso recolector de basura*, el cual reclama la memoria cuando ésta ya no se ocupa.

Tip para prevenir errores 16.1



Cuando los programas de C y C++ no reclaman apropiadamente la memoria asignada en forma dinámica, se produce un error muy común, conocido como fuga de memoria. Las fugas de memoria pueden, de un momento a otro, agotar el suministro de memoria libre y ocasionar que el programa termine antes de tiempo. La recolección de basura automática en Java elimina la vasta mayoría de fugas de memoria; en especial, aquellas que se deben a objetos huérfanos (desreferenciados).

Escribir programas con subprocesamiento múltiple puede ser un proceso complicado. Aunque la mente humana puede realizar funciones de manera concurrente, a las personas se les dificulta saltar de un tren paralelo de pensamiento al otro. Para ver por qué el subprocesamiento múltiple puede ser difícil de programar y comprender, intente realizar el siguiente experimento: Abra tres libros en la página 1 y trate de leerlos concurrentemente. Lea unas cuantas palabras del primer libro; después unas cuantas del segundo y por último otras cuantas del tercero. Luego, regrese a leer las siguientes palabras del primer libro, etcétera. Con este experimento podrá notar los retos que implica el subprocesamiento múltiple: alternar entre los libros, leer brevemente, recordar en dónde se quedó en cada libro, acercar el libro que está leyendo para poder verlo, hacer a un lado los libros que no está leyendo y, entre todo este caos, ¡tratar de comprender el contenido de cada uno!

Aunque Java es tal vez el lenguaje de programación más portable en el mundo, el comportamiento de ejecución de ciertos programas puede variar entre plataformas. En especial, los mecanismos de subprocesos en varios sistemas operativos se encargan de la programación de subprocesos de manera distinta. La programación de subprocesos es el proceso mediante el cual se da cada subproceso al procesador, de manera que el subproceso pueda llevar a cabo su tarea. Cada subproceso tiene una *prioridad* que ayuda al programador de subprocesos a determinar el orden en el que se programan los subprocesos. En algunas plataformas, un subproceso de cierta prioridad se ejecuta hasta completarse o hasta que un subproceso de mayor prioridad requiere el uso del procesador. En el segundo caso, el procesador se asigna al subproceso de mayor prioridad y los subprocesos de menor prioridad deben esperar. En otras plataformas (como Microsoft Windows), los subprocesos se *dividen por tiempo*. En este caso, a cada subproceso se le otorga una cantidad limitada de tiempo (a lo que se le conoce como *cuanto* de tiempo) para ejecutarse en un procesador; cuando ese tiempo expira, el subproceso espera mientras que otro subproceso de igual prioridad obtiene una oportunidad de usar su cuanto de tiempo. Este proceso ocurre en forma cíclica (round-robin), de manera tal que todos los subprocesos de igual prioridad reciben la oportunidad de ejecutarse. Eventualmente, el subproceso original reanuda su ejecución.[2]

2. En la sección 16.3 hablaremos más con más detalle sobre las prioridades y la programación de los subprocesos.

Tip de portabilidad 16.1

La programación de subprocesos en Java es dependiente de la plataforma. Por ende, una aplicación que utiliza subprocesamiento múltiple podría comportarse de manera distinta en diversas implementaciones de Java.

16.2 Estados de los subprocesos: Ciclo de vida de un subproceso

En cualquier momento dado, se dice que un subproceso se encuentra en uno de varios *estados de subproceso* (los cuales se muestran en el diagrama de estados de la figura 16.1). Un nuevo subproceso empieza su ciclo cuando hace la transición al estado *Nacimiento;* permanece en este estado hasta que el programa llama al método `start` de la clase `Thread`, mediante el cual se hace la transición del subproceso al estado *Listo* (algunas veces conocido como el estado *Ejecutable*). Entonces el subproceso que invocó a `start`, el subproceso recientemente iniciado y cualquier otro subproceso en el programa, se ejecutan concurrentemente. Un subproceso hace la transición del estado *Listo* al estado *En ejecución* (es decir, que empieza a ejecutarse) cuando el sistema operativo asigna un procesador al subproceso (lo que también se conoce como *despachar el subproceso*).

Un subproceso *En ejecución* cambia al estado *Muerto* cuando su método `run` se completa, o cuando es terminado por una excepción no atrapada. En el diagrama de estados de UML de la figura 16.1, el estado *Muerto* se representa como el estado final de UML (el símbolo del punto relleno). Cuando un subproceso está *Muerto* y no hay referencias para el objeto del subproceso, el recolector de basura puede eliminar ese objeto del subproceso de la memoria.

Un subproceso cambia al estado *Bloqueado* cuando intenta realizar una tarea que no puede completarse de manera inmediata y debe esperar temporalmente hasta que pueda completarse una tarea; por ejemplo, cuando un subproceso envía una solicitud de entrada/salida. En este caso, el sistema operativo bloquea la ejecución del subproceso hasta que pueda completarse la solicitud de E/S. En ese punto el subproceso cambia al estado *Listo*,

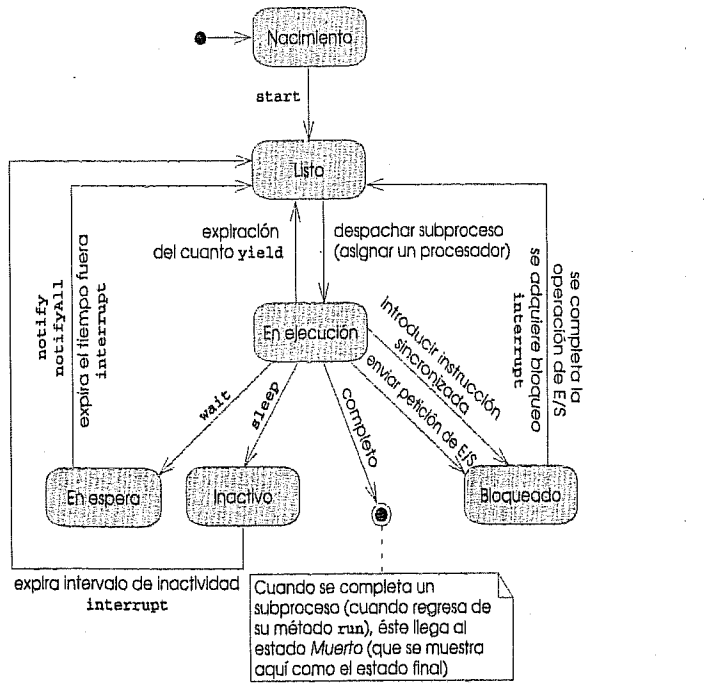


Figura 16.1 Diagrama de estados del ciclo de vida de un subproceso.

http://librosysolucionarios.net

para que pueda despacharse de nuevo y reanudar su ejecución. Un subproceso *Bloqueado* no puede usar un procesador, incluso si hay uno disponible.[3]

Si un subproceso encuentra código que no puede ejecutar todavía (generalmente porque requiere que se cumpla alguna condición), puede llamar al método `wait` de `Object` para cambiar al estado *En espera*. Una vez en este estado, un subproceso cambia al estado *Listo* cuando otro subproceso invoca al método `notify` o `notifyAll` (de la clase `Object`) en el objeto para el cual el subproceso está esperando (es decir, el objeto en el que se llamó a `wait`). El método `notify` hace que un subproceso individual en espera cambie de vuelta al estado *Listo*. Al invocar al método `notifyAll` en un objeto, todos los subprocesos en ese objeto que están en espera cambian de vuelta al estado *Listo*.

Si el programa llama al método `interrupt` (de la clase `Thread`) en un subproceso, se establece la *bandera de interrupción* del subproceso y, dependiendo de su estado, se lanza una excepción `InterruptedException`. Por ejemplo, si un subproceso se encuentra en el estado *Inactivo* y el método `interrupt` se invoca en el subproceso, el método `sleep` lanzará una excepción `InterruptedException`. En este caso, el subproceso sale del estado *Inactivo* y cambia de vuelta al estado *Listo*, para que pueda despacharse de nuevo y procesar la excepción.

Tip de portabilidad 16.2

La notificación en los subprocesos es dependiente de la plataforma. En algunas plataformas, el primer subproceso que espera en un objeto es el primer subproceso notificado. En otras implementaciones se notifica a un subproceso arbitrario que se encuentre en espera en un objeto, sin importar cuál subproceso empezó a estar primero en estado de espera.

Un subproceso *En ejecución* puede llamar al método `sleep` de la clase `Thread` para cambiar al estado *Inactivo*, durante un periodo de milisegundos que se especifican como argumento para `sleep`. Un subproceso inactivo cambia de vuelta al estado *Listo*, cuando expira su tiempo de inactividad designado. Los subprocesos inactivos no pueden usar un procesador, incluso si hay uno disponible. Si el programa llama al método `interrupt` (de la clase `Thread`) en un subproceso inactivo, éste último lanza una excepción `InterruptedException`, sale del estado *Inactivo* y cambia al estado *Listo*, para poder ser despachado nuevamente.

Si un subproceso no puede seguir ejecutándose (lo llamaremos dependiente) a menos que otro subproceso termine, el subproceso dependiente llamará al método `join` del otro subproceso para "unir" los dos subprocesos. Cuando se "unen" dos subprocesos, el que es dependiente sale del estado *En espera* cuando el otro termina su ejecución (cuando entra al estado *Muerto*).

16.3 Prioridades y programación de subprocesos

Los programas en Java pueden utilizar el subprocesamiento múltiple. Todo subproceso en Java tiene una prioridad en el rango entre `Thread.MIN_PRIORITY` (una constante de 1) y `Thread.MAX_PRIORITY` (una constante de 10). De manera informal, los subprocesos con mayor prioridad son más importantes para un programa, y se les debe asignar tiempo del procesador antes que a los subprocesos de menor prioridad. Sin embargo, las prioridades de los subprocesos no pueden garantizar el orden en el que se ejecutarán. De manera predeterminada, a cada subproceso se le asigna la prioridad `Thread.NORM_PRIORITY` (una constante de 5). Cada nuevo subproceso hereda la prioridad del subproceso que lo crea.

La mayoría de las plataformas de Java soportan un concepto conocido como *partición de tiempo (timeslicing)*, el cual permite a los subprocesos de igual prioridad compartir un procesador. Sin la partición de tiempo, cada subproceso en un conjunto de subprocesos de igual prioridad se ejecuta hasta completarse (a menos que salga del estado *En ejecución* y entre al estado *En espera*, *Inactivo* o *Bloqueado;* o que sea interrumpido por un subproceso de mayor prioridad), antes de que otros subprocesos de igual prioridad tengan una oportunidad para ejecutarse. Con la partición de tiempo, cada subproceso recibe un breve lapso del procesador, conocido como *cuanto*, durante el cual puede ejecutarse.

Cuando el cuanto expira, incluso aunque el subproceso no haya terminado de ejecutarse, el procesador se quita de ese subproceso y se asigna al siguiente de igual prioridad, si hay uno disponible.

3. En la sección 16.7 veremos otro caso, en el cual un subproceso entra al estado *Bloqueado*.

El trabajo del *programador de subprocesos* es mantener el subproceso de mayor prioridad *En ejecución* en todo momento y, si hay más de un proceso de mayor prioridad, el programador de subprocesos debe asegurarse que todos los subprocesos se ejecuten durante un cuanto cada uno, en forma cíclica (round-robin). En la figura 16.2 se muestra la cola de prioridades multinivel de Java para los subprocesos. En la figura, suponiendo que hay una computadora con un solo procesador, los subprocesos *A* y *B* se ejecutan cada uno durante un cuanto, en forma cíclica hasta que ambos subprocesos terminan su ejecución. Esto significa que *A* obtiene un cuanto de tiempo para ejecutarse; después *B* obtiene un cuanto. Luego *A* obtiene otro cuanto; y después B, otro. Esto continúa hasta que un subproceso se completa. Entonces, el procesador dedica todo su poder al subproceso restante (a menos que esté listo otro subproceso de esa prioridad). A continuación, el subproceso *C* se ejecuta hasta completarse. Los subprocesos *D*, *E* y *F* se ejecutan cada uno durante un cuanto, en forma cíclica hasta que todos terminan de ejecutarse. Observe que, dependiendo del sistema operativo, los nuevos procesos de mayor prioridad podrían aplazar (tal vez en forma indefinida) la ejecución de los subprocesos de menor prioridad. Comúnmente, a dicho *aplazamiento indefinido* se le conoce, en forma más colorida, como *inanición*. La prioridad de un subproceso puede ajustarse mediante el método `setPriority` de la clase `Thread`, el cual acepta un argumento `int`. Si el argumento no se encuentra en el rango de 1 a 10, `setPriority` lanza una excepción `IllegalArgumentException`. El método `getPriority` de la clase `Thread` devuelve la prioridad del subproceso.

Un subproceso puede llamar al método `yield` de la clase `Thread` para dar a otros una oportunidad para ejecutarse. Sin embargo, siempre que un subproceso de mayor prioridad entra al estado *Listo*, el sistema operativo sustituye al proceso actual para dar preferencia al de mayor prioridad (una operación conocida como *programación preferente*). Por lo tanto, un subproceso no puede ceder (`yield`) a otro de mayor prioridad, ya que el primero hubiera sido desplazado cuando el subproceso de mayor prioridad estuviera listo. Si sólo estuvieran

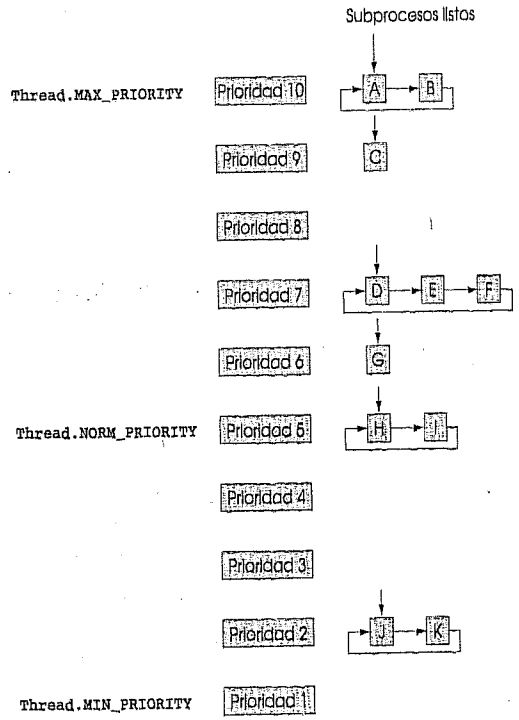


Figura 16.2 Ejemplo de programación de prioridades de subprocesos.

listos subprocesos de menor prioridad al momento de una llamada a `yield`, el subproceso actual sería el de mayor prioridad y continuaría ejecutándose. Por lo tanto, un subproceso cede (`yield`) para dar a los subprocesos de igual prioridad una oportunidad para ejecutarse. En un sistema con partición de tiempo esto es innecesario, ya que los subprocesos de igual prioridad se ejecutarán cada uno durante su cuanto (o hasta que pierdan el procesador por alguna otra razón), y los otros subprocesos de igual prioridad se ejecutarán en forma cíclica. Por lo tanto, `yield` es útil para los sistemas sin partición de tiempo, en donde un subproceso comúnmente se ejecuta hasta terminarse antes de que otro de igual prioridad tenga una oportunidad para ejecutarse.



Tip de rendimiento 16.2

En sistemas sin partición de tiempo, los subprocesos cooperativos de igual prioridad deben llamar periódicamente a `yield`, para permitir a sus iguales proceder sin problemas.



Tip de portabilidad 16.3

Los subprogramas (applets) y aplicaciones de Java deben programarse de manera que puedan funcionar en todas las plataformas de Java, para lograr el objetivo de Java en cuanto a una verdadera portabilidad. Al diseñar subprogramas y aplicaciones que utilicen subprocesos, debe considerar las capacidades de subprocesamiento de todas las plataformas en las que se ejecutarán esos subprogramas y aplicaciones.

Un subproceso se ejecuta hasta que muere; al bloquearse su entrada/salida (o por alguna otra razón); al llamar a `sleep`, `wait` o `yield`; cuando es desplazado por un subproceso de mayor prioridad; o al expirar su cuanto. Un subproceso con mayor prioridad que el subproceso *En ejecución* puede estar listo (y por lo tanto, desplazar al subproceso *En ejecución*) si se inicia dicho subproceso; si dicho subproceso está inactivo y luego se despierta; si la E/S se completa para un subproceso que tenía esa E/S bloqueada; o si se llama al método `notify` o `notifyAll` en un objeto en el que se haya llamado previamente a `wait`.

16.4 Creación y ejecución de subprocesos

En la figura 16.3 se demuestran las técnicas básicas de subprocesamiento como construir objetos `Thread` y utilizar el método `sleep` de `Thread`. El programa crea tres subprocesos de ejecución, cada uno con la prioridad predeterminada `Thread.NORM_PRIORITY`. Cada subproceso muestra un mensaje indicando que va a estar inactivo durante un intervalo aleatorio de 0 a 5000 milisegundos, y luego se vuelve inactivo. Cuando cada subproceso despierta, muestra su nombre, indica que ha dejado de estar inactivo, termina y entra al estado *Muerto*. En este ejemplo podrá ver que el método `main` (es decir, el *subproceso `main` de ejecución*) termina antes de que la aplicación termine. El programa consta de dos clases: `ProbadorSubprocesos` (líneas 4 a 23), la cual crea los tres subprocesos; e `ImprimirSubproceso` (líneas 26 a 59), la cual contiene en su método `run` las acciones que cada objeto `ImprimirSubproceso` llevará a cabo.

```
1  // Fig. 16.3: ProbadorSubprocesos.java
2  // Varios subprocesos imprimiendo en distintos intervalos.
3
4  public class ProbadorSubprocesos {
5
6     public static void main( String [] args )
7     {
8        // crear y nombrar a cada subproceso
9        ImprimirSubproceso subproceso1 = new ImprimirSubproceso( "subproceso1" );
10       ImprimirSubproceso subproceso2 = new ImprimirSubproceso( "subproceso2" );
11       ImprimirSubproceso subproceso3 = new ImprimirSubproceso( "subproceso3" );
12
13       System.err.println( "Iniciando subprocesos" );
14
15       subproceso1.start(); // iniciar subproceso1 y colocarlo en estado listo
16       subproceso2.start(); // iniciar subproceso2 y colocarlo en estado listo
17       subproceso3.start(); // iniciar subproceso3 y colocarlo en estado listo
```

Figura 16.3 Subprocesos inactivos e imprimiendo. (Parte 1 de 3.)

http://librosysolucionarios.net

```
18
19          System.err.println( "Subprocesos iniciados, termina main\n" );
20
21       } // fin de main
22
23    } // fin de la clase ProbadorSubprocesos
24
25    // la clase ImprimirSubproceso controla la ejecución de los subprocesos
26    class ImprimirSubproceso extends Thread {
27       private int tiempoInactividad;
28
29       // asignar nombre a subproceso, llamando al constructor de la superclase
30       public ImprimirSubproceso( String nombre )
31       {
32          super( nombre );
33
34          // elegir tiempo de inactividad aleatorio entre 0 y 5 segundos
35          tiempoInactividad = ( int ) ( Math.random() * 5001 );
36       }
37
38       // el método run es el código a ejecutar por el nuevo subproceso
39       public void run()
40       {
41          // colocar subproceso en inactividad durante el tiempo indicado por tiempoInactividad
42          try {
43             System.err.println(
44                getName() + " va a estar inactivo durante " + tiempoInactividad );
45
46             Thread.sleep( tiempoInactividad );
47          }
48
49          // si el subproceso se interrumpió durante su inactividad, imprimir rastreo de la pila
50          catch ( InterruptedException excepcion ) {
51             excepcion.printStackTrace();
52          }
53
54          // imprimir nombre del subproceso
55          System.err.println( getName() + " termino su inactividad" );
56
57       } // fin del método run
58
59    } // fin de la clase ImprimirSubproceso
```

```
Iniciando subprocesos
Subprocesos iniciados, termina main

subproceso1 va a estar inactivo durante 1217
subproceso2 va a estar inactivo durante 3989
subproceso3 va a estar inactivo durante 662
subproceso3 termino su inactividad
subproceso1 termino su inactividad
subproceso2 termino su inactividad
```

Figura 16.3 Subprocesos inactivos e imprimiendo. (Parte 2 de 3.)

http://librosysolucionarios.net


```
Iniciando subprocesos
subproceso1 va a estar inactivo durante 314
subproceso2 va a estar inactivo durante 1990
Subprocesos iniciados, termina main

subproceso3 va a estar inactivo durante 3016
subproceso1 termino su inactividad
subproceso2 termino su inactividad
subproceso3 termino su inactividad
```

Figura 16.3 Subprocesos inactivos e imprimiendo. (Parte 3 de 3.)

La clase **ImprimirSubproceso** (líneas 26 a 59) extiende a **Thread**, por lo que cada objeto de **ImprimirSubproceso** puede ejecutarse concurrentemente. La clase está compuesta por un campo **tiempoInactividad** (línea 27), un constructor (líneas 30 a 36) y un método **run** (líneas 39 a 57). La variable **tiempoInactividad** almacena un valor entero aleatorio, el cual se elige cuando se hace una llamada al constructor de un nuevo objeto **ImprimirSubproceso**. Cada subproceso controlado por un objeto **ImprimirSubproceso** queda inactivo durante la cantidad de tiempo especificada por el campo **tiempoInactividad** del objeto **ImprimirSubproceso** correspondiente, y después imprime su nombre.

El constructor de **ImprimirSubproceso** (líneas 30 a 36) inicializa a **tiempoInactividad** en un entero aleatorio de 0 a 5000. Cuando a un objeto **ImprimirSubproceso** se le asigna un procesador por primera vez, su método **run** empieza a ejecutarse. En las líneas 43 y 44 se muestra un mensaje indicando el nombre del subproceso que se encuentra en ejecución, e indicando si ese subproceso va a estar inactivo durante cierto número de milisegundos. Observe que en la línea 44 se utiliza el método **getName** del subproceso que se encuentra en ejecución para obtener su nombre, el cual se especificó como un argumento tipo cadena al constructor de **ImprimirSubproceso**. En este punto, el subproceso pierde el procesador y el sistema permite que otro subproceso se ejecute. Cuando el subproceso se despierta, vuelve a entrar al estado *Listo*, en donde espera hasta que el sistema asigna un procesador al subproceso. Cuando el objeto **ImprimirSubproceso** entra de nuevo al estado *En ejecución*, en la línea 55 se muestra el nombre del subproceso en un mensaje que indica que ha dejado de estar inactivo; después el método **run** termina. Esto coloca al subproceso en el estado *Muerto*.

Tip para prevenir errores 16.2



Debe proporcionar un nombre a cada uno de los objetos **Thread**. *Al depurar un programa con subprocesamiento múltiple, estos nombres identifican cuáles subprocesos se están ejecutando.*

El método **main** de **ProbadorSubprocesos** (líneas 6 a 21) crea y nombra a estos tres objetos **ImprimirSubproceso** (líneas 9 a 11). En las líneas 15 a 17 se invoca al método **start** de cada subproceso para pasar los objetos **ImprimirSubproceso** del estado *Nacimiento* al estado *Listo*. El método **start** regresa inmediatamente de cada invocación; después en la línea 19 se imprime un mensaje indicando que se iniciaron los subprocesos. Observe que todo el código en este programa (exceptuando el código en el método **run**) se ejecuta en el subproceso principal. El constructor de **ImprimirSubproceso** también se ejecuta en el subproceso principal, ya que cada objeto **ImprimirSubproceso** se crea en el método **main**. Cuando termina el método **main** (línea 21), el programa continúa ejecutándose ya que aún hay subprocesos vivos (es decir, los subprocesos se iniciaron y no han llegado todavía al estado *Muerto*). El programa no terminará sino hasta que muera su último subproceso. Cuando el sistema asigna un procesador a un objeto **ImprimirSubproceso** por primera vez, el subproceso entra al estado *En ejecución* y el método **run** del subproceso empieza a ejecutarse.

Los resultados de ejemplo para este programa muestran el nombre de cada subproceso y su tiempo de inactividad, cuando el subproceso queda inactivo. El subproceso con el menor tiempo de inactividad es el que generalmente se despierta primero, indica que ha dejado de estar inactivo y termina. En la sección 16.8 hablaremos sobre las cuestiones relacionadas con el subprocesamiento múltiple que podrían evitar que el subproceso con el menor tiempo de inactividad se despertara primero. Observe en el segundo conjunto de resultados de

ejemplo que `subproceso1` y `subproceso2` muestran cada uno su nombre y tiempo de inactividad, antes de que el método `main` haya completado su ejecución. Esta situación demuestra que, una vez que varios procesos se encuentren en el estado *Listo*, cualquier proceso puede asignarse al procesador.

16.5 Sincronización de subprocesos

A menudo, varios subprocesos de ejecución manipulan a un objeto compartido. Si los subprocesos con acceso al objeto compartido simplemente lo leen, entonces no hay necesidad de evitar que éste sea utilizado por más de un subproceso a la vez. Sin embargo, cuando varios subprocesos comparten un objeto, y éste puede ser modificado por uno o más de los subprocesos, pueden ocurrir resultados indeterminados a menos que el objeto compartido se administre de manera apropiada. Si un subproceso se encuentra en el proceso de actualizar el objeto, y otro trata de actualizarlo también, el objeto reflejará la actualización que ocurra después. Si el objeto es un arreglo u otra estructura de datos en la que los subprocesos puedan actualizar partes separadas del objeto concurrentemente, es posible que una parte del objeto refleje la información de un subproceso mientras otra parte refleje la información de un subproceso distinto. Cuando esto ocurre, el comportamiento del programa no puede determinarse. Algunas veces el programa producirá los resultados correctos; sin embargo, en otras ocasiones producirá los resultados incorrectos. En cualquier caso, no habrá un mensaje de error para indicar que el objeto compartido se manipuló en forma incorrecta.

El problema puede resolverse si se da a un subproceso a la vez el acceso exclusivo al código que manipula al objeto compartido. Durante ese tiempo, otros subprocesos que deseen manipular el objeto deben mantenerse en espera. Cuando el subproceso con acceso exclusivo al objeto termina de manipularlo, a uno de los subprocesos que estaba en espera se le debe permitir que continúe ejecutándose. De este modo, cada subproceso que utiliza el objeto compartido excluye a los demás subprocesos de hacerlo en forma simultánea. A esto se le conoce como *exclusión mutua* o *sincronización de subprocesos*.

Java utiliza *monitores* para llevar a cabo la sincronización. Cada objeto tiene un monitor que permite que se ejecute un subproceso a la vez, dentro de una *instrucción sincronizada* en el objeto. Para lograr esto, se *bloquea* el objeto cuando el programa llega a la instrucción sincronizada (lo que también se conoce como *obtener el bloqueo*). Si hay varias instrucciones sincronizadas tratando de sincronizarse en el objeto al mismo tiempo, sólo una instrucción (`statement`) sincronizada puede estar activa en un objeto a la vez; todos los demás subprocesos que intenten entrar a una instrucción sincronizada en el mismo objeto se colocan en el estado *Bloqueado*. Cuando una instrucción sincronizada termina de ejecutarse, se libera el bloqueo en ese objeto y el monitor permite que continúe el subproceso bloqueado con la mayor prioridad, que esté tratando de entrar a una instrucción sincronizada. Java también permite el uso de *métodos sincronizados*, los cuales se utilizan en los ejemplos de las secciones 16.7 y 16.8. Un método sincronizado es equivalente a una instrucción sincronizada que encierra al cuerpo completo del método.

Una vez que un subproceso obtiene el bloqueo en un objeto, si el subproceso determina que no puede continuar con su tarea en ese objeto hasta que se cumpla cierta condición, el subproceso puede llamar al método `wait` de `Object` para dejar de ocupar el procesador y liberar el bloqueo en ese objeto. El subproceso libera el bloqueo en el objeto y queda en el estado *En espera* mientras los demás subprocesos tratan de entrar a la(s) instrucción(es) sincronizada(s) en el objeto. Cuando un subproceso que ejecuta una instrucción sincronizada se completa o cumple con la condición en la que otro subproceso podría estar en espera, el subproceso puede llamar al método `notify` de `Object` para permitir a un subproceso en espera cambiar de nuevo al estado *Listo*. En este punto, el subproceso que cambió del estado *En espera* a *Listo* puede tratar de readquirir el bloqueo en el objeto. Aun si el subproceso puede readquirir el bloqueo, tal vez no pueda realizar su tarea en este punto; en cuyo caso el subproceso volverá a entrar al estado *En espera* y liberará el bloqueo. Si un subproceso llama a `notifyAll`, entonces todos los subprocesos en espera del objeto monitor tendrán la oportunidad de readquirir el bloqueo (es decir, todos cambian al estado *Listo*). Recuerde que sólo uno de esos subprocesos puede, en un momento dado, obtener el bloqueo en el objeto; otros subprocesos que traten de adquirir el mismo bloqueo serán bloqueados por el sistema operativo hasta que ese bloqueo esté disponible otra vez (es decir, hasta que ningún otro subproceso se esté ejecutando en una instrucción sincronizada en ese objeto). Todas las clases heredan los métodos `wait`, `notify` y `notifyAll` de la clase `Object`. Cada objeto tiene un monitor.



Error común de programación 16.1

El interbloqueo ocurre cuando un subproceso en espera (al que llamaremos subproceso1) no puede continuar, ya que está esperando a que otro subproceso (al que llamaremos subproceso2) continúe. De manera similar, el subproceso2 no puede continuar porque está esperando a que el subproceso1 continúe. Los dos subprocesos se están esperando mutuamente; por lo tanto, las acciones que permitirían que cada subproceso continuara su ejecución nunca ocurrirán.



Tip para prevenir errores 16.3

Cuando varios subprocesos manipulan a un objeto compartido mediante el uso de monitores, asegúrese que si un subproceso llama al método `wait` *para entrar al estado* En espera *por el objeto compartido, haya un subproceso separado que en algún momento llame al método* `notify` *para cambiar el subproceso, que espera el objeto compartido, de vuelta al estado* Listo. *Si puede haber varios subprocesos esperando el objeto compartido, un subproceso separado puede llamar al método* `notifyAll` *como precaución para asegurar que todos los subprocesos en espera tengan otra oportunidad para realizar sus tareas.*



Tip de rendimiento 16.3

La sincronización para lograr la exactitud en los programas con subprocesamiento múltiple puede hacer que los programas se ejecuten con más lentitud, como resultado de una sobrecarga del monitor y la transición frecuente de los subprocesos entre los estados En ejecución, En espera y Listo. *Sin embargo, ¡no hay mucho que decir en cuanto a los programas con subprocesamiento múltiple incorrectos y altamente eficientes!*



Observación de ingeniería de software 16.2

El bloqueo que ocurre con la ejecución de métodos `synchronized` *podría conducir a un interbloqueo, si los bloqueos nunca se liberan. Cuando ocurren excepciones, el mecanismo de excepciones de Java se coordina con el mecanismo de sincronización para liberar bloqueos y evitar estos tipos de interbloqueo.*



Error común de programación 16.2

Es un error si un subproceso ejecuta un método `wait`, `notify` *o* `notifyAll` *en un objeto, sin haber adquirido un bloqueo para ese objeto. Esto produce una excepción* `IllegalMonitorStateException`.

16.6 Relación productor/consumidor sin sincronización

En una *relación productor/consumidor*, la parte de una aplicación correspondiente al *productor* genera datos y los almacena en un objeto compartido; y la parte de una aplicación correspondiente al *consumidor* lee los datos del objeto compartido. En una relación productor/consumidor con subprocesamiento múltiple, un *subproceso productor* genera datos y los coloca en un objeto compartido, conocido como *búfer*. Un *subproceso consumidor* lee los datos del búfer. Si el productor que espera a colocar los siguientes datos en el búfer examina una condición y determina que el consumidor no ha leído todavía los datos anteriores del búfer, el subproceso productor debe llamar a `wait` para que el consumidor tenga oportunidad de leer los datos antes de que éstos se actualicen; de no ser así, el consumidor nunca verá los datos anteriores y la aplicación los perderá. Cuando el subproceso consumidor lee los datos, debe llamar a `notify` para permitir que un productor en espera almacene el siguiente valor. Si un subproceso consumidor encuentra el búfer vacío, o descubre que los datos anteriores ya han sido leídos, debe llamar a `wait`; de no ser así, el consumidor podría leer datos obsoletos del búfer. Cuando el productor coloca los siguientes datos en el búfer, debe llamar a `notify` para permitir que el subproceso consumidor continúe, de manera que pueda leer los nuevos datos.

Consideremos ahora cómo pueden surgir errores lógicos si no sincronizamos el acceso entre varios subprocesos que manipulan datos compartidos. Nuestro siguiente ejemplo (figuras 16.4 a 16.8) implementa una relación productor/consumidor, en la que un subproceso productor escribe una secuencia de números (utilizamos del 1 al 4) en un búfer compartido: una ubicación de memoria compartida entre dos subprocesos (una variable `int` llamada `bufer` en la figura 16.7 de este ejemplo). El subproceso consumidor lee estos datos del búfer compartido y los muestra en pantalla. La salida del programa muestra los valores que el productor escribe (produce) en el búfer compartido y los valores que el consumidor lee (consume) del búfer compartido.

Cada valor que el subproceso productor escribe en el búfer compartido debe ser consumido exactamente una vez por el subproceso consumidor. Sin embargo, los subprocesos en este ejemplo no están sincronizados. Por ende, los datos pueden perderse si el productor coloca nuevos datos en el búfer compartido antes de que el consumidor consuma los datos anteriores. Además, los datos pueden duplicarse de manera incorrecta si el con-

http://librosysolucionarios.net

```
1  // Fig. 16.4: Bufer.java
2  // La interfaz Bufer especifica los métodos llamados por el Productor y el Consumidor.
3
4  public interface Bufer {
5     public void establecer( int valor );  // colocar valor en Bufer
6     public int obtener();                 // devolver valor de Bufer
7  }
```

Figura 16.4 Interfaz `Bufer` utilizada en los ejemplos de productor/consumidor.

sumidor consume los datos otra vez antes de que el productor produzca el siguiente valor. Para mostrar estas posibilidades, el subproceso consumidor del siguiente ejemplo mantiene un total de todos los valores que lee. El subproceso productor produce valores de 1 a 4. Si el consumidor lee sólo una vez cada uno de los valores producidos, el total será de 10. No obstante, si usted ejecuta este programa varias veces, podrá ver que en muy pocas ocasiones, si acaso hay una, el total es 10. Además, para recalcar nuestro punto, los subprocesos productor y consumidor en el ejemplo quedan inactivos durante intervalos aleatorios de hasta tres segundos al estar realizando sus tareas, simulando algún proceso extenso como esperar la entrada del usuario. Por lo tanto, no conocemos exactamente el momento en que el subproceso productor tratará de escribir un nuevo valor, ni conocemos el momento en que el consumidor tratará de leer un valor.

El programa consiste de la interfaz `Bufer` (figura 16.4) y cuatro clases: `Productor` (figura 16.5), `Consumidor` (figura 16.6), `BuferNoSincronizado` (figura 16.7) y `PruebaBuferCompartido` (figura 16.8). La interfaz `Bufer` declara los métodos `establecer` y `obtener` que un `Bufer` debe implementar para permitir a un subproceso `Productor` colocar un valor en el `Bufer`, y para permitir al subproceso `Consumidor` recuperar un valor del `Bufer`, respectivamente. En la figura 16.7 podemos ver la implementación de esta interfaz.

La clase `Productor` (figura 16.5) (una subclase de `Thread`) contiene el campo `ubicacionCompartida` (línea 6), un constructor (líneas 9 a 13) y un método `run` (líneas 16 a 36). El constructor inicializa la referencia `Bufer` llamada `ubicacionCompartida` con un objeto que implementa a la interfaz `Bufer` creada en `main` (figura 16.8), la cual se pasa al constructor como el parámetro `compartido`. Como veremos, éste es un objeto `BuferNoSincronizado` que implementa a la interfaz `Bufer` sin sincronizar el acceso al objeto compartido. El subproceso productor en este programa ejecuta las tareas especificadas en el método `run` (líneas 16 a 36). El ciclo de las líneas 18 a 31 itera cuatro veces. Cada iteración del ciclo invoca al método `sleep` de `Thread` (línea 22) para colocar el subproceso productor en el estado *Inactivo* durante un intervalo aleatorio de tiempo, entre 0 y 3 segundos (para simular un proceso extenso). Cuando el subproceso despierta, en la línea 23 se pasa el valor de la variable de control `cuenta` al método `establecer` del objeto `Bufer`, para establecer el valor del búfer compartido. Cuando el ciclo completa, en las líneas 33 y 34 se muestra un mensaje en la ventana de la consola, indicando que el subproceso terminó de producir datos y que va a terminar. A continuación, el método `run` termina y el subproceso productor entra al estado *Muerto*. Es importante observar que cualquier método que es llamado desde el método `run` (como el método `establecer` de `Bufer`) se ejecuta como parte de ese subproceso de ejecución. De hecho, cada subproceso tiene su propia pila de llamadas a métodos. Este hecho cobrará importancia en la sección 16.7, cuando agreguemos sincronización a la relación productor/consumidor.

```
1  // Fig. 16.5: Productor.java
2  // El método run de Productor controla un subproceso que
3  // almacena los valores de 1 a 5 en ubicacionCompartida.
4
5  public class Productor extends Thread {
6     private Bufer ubicacionCompartida; // referencia al objeto compartido
7
8     // constructor
9     public Productor( Bufer compartido )
10    {
11       super( "Productor" );
12       ubicacionCompartida = compartido;
13    }
14
15    // almacenar valores de 1 a 4 en ubicacionCompartida
16    public void run()
17    {
18       for ( int cuenta = 1; cuenta <= 4; cuenta++ ) {
19
20          // estar inactivo de 0 a 3 segundos y luego colocar valor en Bufer
21          try {
22             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
23             ubicacionCompartida.establecer( cuenta );
24          }
25
26          // si se interrumpió el subproceso inactivo, imprimir rastreo de pila
27          catch ( InterruptedException excepcion ) {
28             excepcion.printStackTrace();
29          }
30
31       } // fin de instrucción for
32
33       System.err.println( getName() + " termino de producir." +
34          "\nTerminando " + getName() + "." );
35
36    } // fin del método run
37
38 } // fin de la clase Productor
```

Figura 16.5 `Productor` representa al subproceso productor en una relación productor/consumidor. (Parte 1 de 2)

Figura 16.5 `Productor` representa al subproceso productor en una relación productor/consumidor. (Parte 2 de 2.)

La clase `Consumidor` (figura 16.6) contiene la variable de instancia `ubicacionCompartida` (línea 6), un constructor (líneas 9 a 13) y un método `run` (líneas 16 a 37). El constructor inicializa la referencia `Bufer` llamada `ubicacionCompartida` con un objeto que implementa a la interfaz `Bufer` creada en `main` (figura 16.8), la cual se pasa al constructor como el parámetro `compartido`. Como veremos, éste es el mismo objeto `BuferNoSincronizado` que se utiliza para inicializar al objeto `Productor`; por lo tanto, los dos subprocesos comparten el mismo objeto. El subproceso consumidor en este programa ejecuta las tareas especificadas en el método `run` (líneas 16 a 37). El ciclo de las líneas 20 a 32 itera cuatro veces. Cada iteración del ciclo invoca al método `sleep` de `Thread` (línea 24) para colocar el subproceso consumidor en el estado *Inactivo* durante un intervalo aleatorio de tiempo, entre 0 y 3 segundos. A continuación, en la línea 25 se utiliza el método `obtener` de `Bufer` para recuperar el valor en el búfer compartido, y se agrega el valor a la variable `suma`. Cuando el ciclo termina, en las líneas 34 a 35 se muestra una línea en la ventana de consola, indicando la suma de los valores consumidos. Después el método `run` termina y coloca al subproceso consumidor en el estado *Muerto*. Una vez que ambos subprocesos entran al estado *Muerto*, el programa termina.

(*Nota:* Utilizamos el método `sleep` en el método `run` de las clases `Productor` y `Consumidor` para enfatizar el hecho de que, en las aplicaciones con subprocesamiento múltiple, no está claro cuándo va a realizar su tarea cada subproceso, ni por cuánto tiempo realizará esa tarea cuando tenga el procesador. Por lo general, estas cuestiones relacionadas con la programación de subprocesos corresponden al sistema operativo de la computadora. En este programa, las tareas de nuestro subproceso son bastante simples: para el productor, iterar cuatro veces y ejecutar una instrucción de asignación; para el consumidor, iterar cuatro veces y agregar un

http://librosysolucionarios.net

```
1  // Fig. 16.6: Consumidor.java
2  // El método run de Consumidor controla un subproceso que itera cuatro
3  // veces y lee un valor de ubicacionCompartida cada vez.
4
5  public class Consumidor extends Thread {
6     private Bufer ubicacionCompartida; // referencia al objeto compartido
7
8     // constructor
9     public Consumidor( Bufer compartido )
10    {
11       super( "Consumidor" );
12       ubicacionCompartida = compartido;
13    }
14
15    // leer el valor de ubicacionCompartida cuatro veces y sumar los valores
16    public void run()
17    {
18       int suma = 0;
19
20       for ( int cuenta = 1; cuenta <= 4; cuenta++ ) {
21
22          // estar inactivo de 0 a 3 segundos, leer un valor de Bufer y agregarlo a suma
23          try {
24             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
25             suma += ubicacionCompartida.obtener();
26          }
27
28          // si se interrumpió el subproceso inactivo, imprimir rastreo de la pila
29          catch ( InterruptedException excepcion ) {
30             excepcion.printStackTrace();
31          }
32       }
33
34       System.err.println( getName() + " leyo valores, dando un total de: " + suma +
35          ".\nTerminando " + getName() + ".");
36
37    } // fin del método run
38
39 } // fin de la clase Consumidor
```

Figura 16.6 **Consumidor** representa el subproceso consumidor en una relación productor/consumidor.

valor a la variable **suma**. Sin la llamada al método **sleep**, y si el productor se ejecuta primero, éste muy probablemente completaría su tarea antes de que el consumidor tuviera la oportunidad de ejecutarse. Si el consumidor se ejecuta primero, muy probablemente consumiría **-1** cuatro veces y después terminaría antes que el productor pudiera producir el primer valor real.)

La clase **BuferNoSincronizado** (figura 16.7) implementa a la interfaz **Bufer** (línea 4) y declara el tipo de objeto compartido entre el **Productor** y el **Consumidor**. En la línea 5 se declara la variable de instancia **bufer** y se inicializa con el valor **-1**. Este valor se utiliza para demostrar el caso en el que el **Consumidor** intenta consumir un valor antes que el **Productor** coloque un valor en **bufer**. Los métodos **establecer** (líneas 8 a 14) y **obtener** (líneas 17 a 23) no sincronizan el acceso al campo **bufer**. El método **set** simplemente asigna su argumento a **bufer** (línea 13) y el método **obtener** lo único que hace es devolver el valor de **bufer** (línea 22). Observe que cada uno de los métodos utiliza el método estático **currentThread** de **Thread** para obtener una referencia al subproceso que está actualmente en ejecución, y después utiliza el método **getName** de ese subproceso para obtener su nombre y mostrarlo en pantalla.


```
1  // Fig. 16.7: BuferNoSincronizado.java
2  // BuferNoSincronizado representa a un solo entero compartido.
3
4  public class BuferNoSincronizado implements Bufer {
5     private int bufer = -1; // compartido por los subprocesos productor y consumidor
6
7     // colocar valor en bufer
8     public void establecer( int valor )
9     {
10       System.err.println( Thread.currentThread().getName() +
11          " escribe " + valor );
12
13       bufer = valor;
14    }
15
16    // devolver valor de bufer
17    public int obtener()
18    {
19       System.err.println( Thread.currentThread().getName() +
20          " lee " + bufer );
21
22       return bufer;
23    }
24
25 } // fin de la clase BuferNoSincronizado
```

Figura 16.7 **BuferNoSincronizado** mantiene el entero compartido que es utilizado por un subproceso productor y un subproceso consumidor mediante los métodos **establecer** y **obtener**.

La clase **PruebaBuferCompartido** contiene el método **main** (líneas 6 a 18), el cual inicia la aplicación. En la línea 9 se crea una instancia de un objeto **BuferNoSincronizado** compartido, y se asigna a la referencia **Bufer** llamada **ubicacionCompartida**. Este objeto almacena los datos que se compartirán entre los subprocesos productor y consumidor. En las líneas 12 a 13 se crea un objeto **Productor** y un objeto **Consumidor**. Cada llamada al constructor pasa el valor de **ubicacionCompartida** como argumento para el constructor, por lo que cada objeto se inicializa con una referencia al mismo objeto **Bufer**. A continuación, en las líneas 15 y 16 se invoca el método **start** en los subprocesos **productor** y **consumidor** para colocarlos en el estado *Listo*. Este método inicia a estos subprocesos y establece la llamada inicial al método **run** de cada subproceso, el cual empezará a ejecutarse cuando a cada subproceso se le asigne un procesador. Finalmente, el método **main** termina y el subproceso **main** de ejecución muere.

```
1  // Fig. 16.8: PruebaBuferCompartido.java
2  // PruebaBuferCompartido crea los subprocesos productor y consumidor.
3
4  public class PruebaBuferCompartido {
5
6     public static void main( String [] args )
7     {
8        // crear el objeto compartido utilizado por los subprocesos
9        Bufer ubicacionCompartida = new BuferNoSincronizado();
10
11       // crear objetos productor y consumidor
12       Productor productor = new Productor( ubicacionCompartida );
```

Figura 16.8 Subprocesos que modifican a un objeto compartido sin sincronización. (Parte 1 de 2.)

http://librosysolucionarios.net

```
13          Consumidor consumidor = new Consumidor( ubicacionCompartida );
14
15          productor.start(); // iniciar subproceso productor
16          consumidor.start(); // iniciar subproceso consumidor
17
18       } // fin de main
19
20    } // fin de la clase PruebaBuferCompartido
```

```
Consumidor lee -1
Productor escribe 1
Consumidor lee 1
Consumidor lee 1
Consumidor lee 1
Consumidor leyo valores, dando un total de: 2.
Terminando Consumidor.
Productor escribe 2
Productor escribe 3
Productor escribe 4
Productor termino de producir.
Terminando Productor.
```

```
Productor escribe 1
Productor escribe 2
Consumidor lee 2
Productor escribe 3
Consumidor lee 3
Productor escribe 4
Productor termino de producir.
Terminando Productor.
Consumidor lee 4
Consumidor lee 4
Consumidor leyo valores, dando un total de: 13.
Terminando Consumidor.
```

```
Productor escribe 1
Consumidor lee 1
Productor escribe 2
Consumidor lee 2
Productor escribe 3
Consumidor lee 3
Productor escribe 4
Productor termino de producir.
Terminando Productor.
Consumidor lee 4
Consumidor leyo valores, dando un total de: 10.
Terminando Consumidor.
```

Figura 16.8 Subprocesos que modifican a un objeto compartido sin sincronización. (Parte 2 de 2.)

Recuerde que en la descripción general de este ejemplo vimos que nos gustaría que el subproceso `Productor` se ejecutara primero, y que el consumidor consumiera cada valor producido por el productor exactamente una vez. No obstante, al estudiar el primer conjunto de resultados de la figura 16.8 podemos ver que el consumidor recuperó un valor (–1) antes de que el productor colocara un valor en el búfer, y que el valor 1 se consumió tres veces. El consumidor terminó su ejecución antes que el productor tuviera la oportunidad de producir los valores `2`, `3` y `4`. Por lo tanto, esos tres valores se perdieron. En el segundo conjunto de resultados podemos ver que se perdió el valor `1`, ya que los valores `1` y `2` se produjeron antes que el subproceso consumidor pudiera leer el valor `1`. Además, el valor `4` se consumió dos veces. El último conjunto de resultados de ejemplo demuestra que es posible, con algo de suerte, obtener los resultados apropiados en donde cada valor que produce el productor es consumido sólo una vez por el consumidor, pero este comportamiento no puede garantizarse. Este ejemplo muestra con claridad que el acceso a un objeto compartido por subprocesos concurrentes debe controlarse cuidadosamente; de no ser así, un programa podría producir resultados incorrectos.

Para resolver los problemas de datos perdidos y de datos consumidos más de una vez, en la sección 16.7 presentamos un ejemplo en el que utilizamos la palabra clave `synchronized`, junto con los métodos `wait` y `notify` de `Object`, para sincronizar el acceso al código que manipula al objeto compartido. Cuando un método declarado como `synchronized` se ejecuta en un objeto, este objeto se *bloquea* de manera que ningún otro método `synchronized` pueda ejecutarse en ese objeto al mismo tiempo.

16.7 Relación productor/consumidor con sincronización

La aplicación de las figuras 16.9 y 16.10 demuestra cómo un productor y un consumidor utilizan un búfer compartido mediante la sincronización. En este caso, el consumidor consume sólo hasta que el productor produce el valor, y el productor produce un nuevo valor sólo hasta que el consumidor consume el valor anterior producido. En este ejemplo, reutilizamos la interfaz `Bufer` (figura 16.4) junto con las clases `Productor` (figura 16.5) y `Consumidor` (figura 16.6) del ejemplo de la sección 16.6. Este método nos permite demostrar que los subprocesos que utilizan el objeto compartido no saben que están siendo sincronizados. El código que lleva a cabo la sincronización se coloca en los métodos `establecer` y `obtener` de la clase `BuferSincronizado` (figura 16.9), la cual implementa a la interfaz `Bufer` (línea 4). Por lo tanto, los métodos `run` de `Productor` y `Consumidor` simplemente llaman a los métodos `establecer` y `obtener` del objeto compartido, como en el ejemplo de la sección 16.6.

La clase `BuferSincronizado` (figura 16.9) contiene dos campos: `bufer` (línea 5) y `cuentaBuferOcupado` (línea 6). Los métodos `set` (líneas 9 a 41) y `get` (líneas 44 a 76) son ahora métodos `synchronized`; por lo tanto, sólo un subproceso puede llamar a uno de estos métodos a la vez, en un objeto `BuferSincronizado` específico. El campo `cuentaBuferOcupado` se conoce como una *variable de condición*; los métodos utilizan esta variable `int` en expresiones condicionales para determinar si es turno del productor o del consumidor para realizar una tarea. Si `cuentaBuferOcupado` es igual a cero, el búfer está vacío y el productor puede llamar al método `establecer` para colocar un valor en la variable `bufer`. Esta condición también significa que el consumidor no puede llamar al método `obtener` de `BuferSincronizado` para leer el valor de `bufer` (porque, como se dijo anteriormente, está vacío). Si `cuentaBuferOcupado` es igual a uno, el consumidor puede llamar al método `obtener` de `BuferSincronizado` para leer un valor de la variable `bufer`, ya que esta variable contiene nueva información. Esta condición también significa que el productor no puede llamar al método `establecer` de `BuferSincronizado` para colocar un valor en `bufer`, porque en esos momentos se encuentra lleno.

```
1  // Fig. 16.9: BuferSincronizado.java
2  // BuferSincronizado sincroniza el acceso a un solo entero compartido.
3
4  public class BuferSincronizado implements Bufer {
5     private int bufer = -1; // compartido por los subprocesos productor y consumidor
6     private int cuentaBuferOcupado = 0; // cuenta de búferes ocupados
7
```

Figura 16.9 `BuferSincronizado` sincroniza el acceso a un entero compartido. (Parte 1 de 3.)

http://librosysolucionarios.net

```
8     // colocar valor en búfer
9     public synchronized void establecer( int valor )
10    {
11       // obtener nombre del subproceso que llamó a este método, para mostrarlo en pantalla
12       String nombre = Thread.currentThread().getName();
13
14       // mientras no haya ubicaciones vacías, colocar subproceso en estado de espera
15       while ( cuentaBuferOcupado >= 1 ) {
16
17          // mostrar información del subproceso y del búfer, después esperar
18          try {
19             System.err.println( nombre + " trata de escribir." );
20             mostrarEstado( "Bufer lleno. " + nombre + " espera." );
21             wait();
22          }
23
24          // si se interrumpió el subproceso en espera, imprimir el rastreo de la pila
25          catch ( InterruptedException excepcion ) {
26             excepcion.printStackTrace();
27          }
28
29       } // fin de instrucción while
30
31       bufer = valor; // establecer nuevo valor de búfer
32
33       // indicar que el productor no puede almacenar otro valor
34       // sino hasta que el consumidor recupere el valor actual del búfer
35       ++cuentaBuferOcupado;
36
37       mostrarEstado( nombre + " escribe " + bufer );
38
39       notify(); // indicar al subproceso en espera que entre al estado listo
40
41    } // fin del método establecer; se libera el bloqueo en BuferSincronizado
42
43    // devolver valor de bufer
44    public synchronized int obtener()
45    {
46       // obtener nombre del subproceso que llamó a este método, para mostrarlo en pantalla
47       String nombre = Thread.currentThread().getName();
48
49       // mientras no haya datos que leer, colocar subproceso en estado de espera
50       while ( cuentaBuferOcupado == 0 ) {
51
52          // mostrar información del subproceso y del búfer, después esperar
53          try {
54             System.err.println( nombre + " trata de leer." );
55             mostrarEstado( "Bufer vacio. " + nombre + " espera." );
56             wait();
57          }
58
59          // si se interrumpió el subproceso en espera, imprimir el rastreo de la pila
60          catch ( InterruptedException excepcion ) {
61             excepcion.printStackTrace();
```

Figura 16.9 `BuferSincronizado` sincroniza el acceso a un entero compartido. (Parte 2 de 3.)


```
62          }
63
64       } // fin de instrucción while
65
66       // indicar que el productor puede almacenar otro valor
67       // ya que el consumidor acaba de recuperar el valor de bufer
68       --cuentaBuferOcupado;
69
70       mostrarEstado( nombre + " lee " + bufer );
71
72       notify(); // indicar al subproceso en espera que esté listo para ejecutarse
73
74       return bufer;
75
76    } // fin del método obtener; libera bloqueo en BuferSincronizado
77
78    // mostrar la operación actual y el estado del búfer
79    public void mostrarEstado( String operacion )
80    {
81       StringBuffer lineaSalida = new StringBuffer( operacion );
82       lineaSalida.setLength( 40 );
83       lineaSalida.append( bufer + "\t\t" + cuentaBuferOcupado );
84       System.err.println( lineaSalida );
85       System.err.println();
86    }
87
88 } // fin de la clase BuferSincronizado
```

Figura 16.9 `BuferSincronizado` sincroniza el acceso a un entero compartido. (Parte 3 de 3.)

Cuando el método `run` del subproceso `Productor` invoca al método sincronizado `establecer` (de la línea 23 de la figura 16.5), el subproceso trata de adquirir un bloqueo en el objeto monitor `BuferSincronizado`. Si el bloqueo está disponible, el subproceso `Productor` adquiere el bloqueo. Entonces el ciclo `while` de las líneas 15 a 29 determina si `cuentaBuferOcupado` es igual a uno. De ser así, significa que el búfer está lleno, por lo que en la línea 19 se muestra un mensaje indicando que el subproceso `Productor` está tratando de escribir un valor, y en la línea 20 se invoca al método `mostrarEstado` (líneas 79 a 86) para mostrar otro mensaje indicando que el búfer está lleno y que el subproceso `Productor` se encuentra en el estado *En espera*. En la línea 21 se invoca al método `wait` (heredado de `Object` por `BuferSincronizado`) para colocar el subproceso que llamó al método `establecer` (es decir, el subproceso `Productor`) en el estado *En espera* por el objeto `BuferSincronizado`. La llamada a `wait` ocasiona que el subproceso que hace la llamada libere el bloqueo en el objeto `BuferSincronizado`. Esto es importante, ya que el subproceso no puede realizar su tarea en ese momento, y porque se debe permitir a otros subprocesos tener acceso al objeto en este momento, para permitir que cambie la condición (`cuentaBuferOcupado == 1`). Ahora, otro subproceso puede tratar de adquirir el bloqueo del objeto `BuferSincronizado` e invocar a los métodos `establecer` u `obtener` del objeto.

El subproceso productor permanece en el estado *En espera* hasta que el subproceso es notificado por otro subproceso que puede proceder; en este punto el subproceso productor regresa al estado *Listo* y espera a que el sistema asigne un procesador al subproceso. Cuando el subproceso productor regresa al estado *En ejecución*, el subproceso intenta implícitamente readquirir el bloqueo sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso productor lo readquiere y el método `establecer` continúa ejecutándose con la siguiente instrucción después de `wait`. Como `wait` es llamado en un ciclo (líneas 15 a 29), la condición de continuación de ciclo se evalúa de nuevo para determinar si el subproceso puede proceder con su ejecución. De no ser así, `wait` es invocado de nuevo; en caso contrario, el método `establecer` continúa con la siguiente instrucción después del ciclo.

http://librosysolucionarios.net

Observación de ingeniería de software 16.3

Siempre se debe invocar al método `wait` *en un ciclo que evalúe una variable de condición apropiada. Si la condición indica que el subproceso puede realizar su tarea, esto asegura que el subproceso no esperará innecesariamente. Cuando un subproceso en espera es notificado para volver a entrar al estado* Listo, *el subproceso verificará la variable de condición de ciclo antes de continuar con su ejecución. Esto asegura que el subproceso no se vaya a ejecutar de manera errónea, en caso de que haya sido notificado antes de que se cumpla la condición, o si el subproceso fue despertado por un método* `notifyAll` *que estaba destinado a otro subproceso.*

En la línea 31, en el método `establecer` se asigna `valor` a `bufer`. En la línea 35 se incrementa `cuentaBuferOcupado` para indicar que el `bufer` ahora contiene un valor (es decir, un consumidor puede leer un valor y un productor no puede colocar todavía otro valor ahí). En la línea 37 se invoca al método `mostrarEstado` para mostrar una línea en la ventana de consola, indicando que el productor va a escribir un nuevo valor en el `bufer`. En la línea 39 se invoca al método `notify` (heredado de `Object`). Si hay algún subproceso en espera, el primero de ellos entra al estado *Listo*, indicando que el subproceso ahora puede tratar de realizar su tarea otra vez (tan pronto como se le asigne un procesador al subproceso). El método `notify` regresa de inmediato, y el método `establecer` regresa al método que lo llamó.[4] Cuando el método `establecer` regresa, libera implícitamente el bloqueo en la memoria compartida.

Error común de programación 16.3

Olvidar notificar a un subproceso (mediante `notify`*) que está en espera de una condición es un error lógico. El subproceso permanecerá en el estado* En espera, *con lo cual se evita que el subproceso realice cualquier trabajo adicional. Dicha espera puede llevar a un aplazamiento indefinido o interbloqueo.*

Los métodos `establecer` y `obtener` se implementan de manera similar. Cuando el método `run` del subproceso `Consumidor` invoca al método sincronizado `obtener` (de la línea 25 de la figura 16.6), el subproceso trata de adquirir un bloqueo en el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso `Consumidor` adquiere el bloqueo. Después, el ciclo while de las líneas 50 a 64 determina si `cuentaBuferOcupado` es igual a 0. De ser así, significa que el búfer está vacío, por lo que en la línea 54 se muestra un mensaje indicando que el subproceso `Consumidor` está tratando de leer un valor, y en la línea 55 se invoca al método `mostrarEstado` para mostrar otro mensaje indicando que el búfer está vacío y que el subproceso `Consumidor` está esperando. En la línea 56 se invoca al método `wait` para colocar el subproceso que llamó al método `establecer` (es decir, el subproceso `Consumidor`) en el estado *En espera* por el objeto `BuferSincronizado`. De nuevo, la llamada a `wait` ocasiona que el subproceso que hizo la llamada libere el bloqueo sobre el objeto `BuferSincronizado`, para que otro subproceso pueda tratar de adquirir el bloqueo sobre este objeto, e invocar a su método `establecer` u `obtener`. Si el bloqueo en el objeto `BuferSincronizado` no está disponible (por ejemplo, si el subproceso `Productor` no ha regresado todavía del método `establecer`), el subproceso `Consumidor` se bloquea hasta que el bloqueo sobre el objeto esté disponible.

El subproceso consumidor permanece en el estado *En espera* hasta que el subproceso es notificado por otro subproceso que puede proceder; en este punto el subproceso consumidor regresa al estado *Listo* y espera a que el sistema asigne un procesador al subproceso. Cuando el subproceso regresa al estado *En ejecución*, intenta implícitamente readquirir el bloqueo sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso consumidor lo readquiere y el método `obtener` continúa ejecutándose con la siguiente instrucción después de `wait`. Como `wait` es llamado en un ciclo (líneas 50 a 64), la condición de continuación de ciclo se evalúa de nuevo para determinar si el subproceso puede proceder con su ejecución. De no ser así, `wait` es invocado de nuevo; en caso contrario, el método `obtener` continúa con la siguiente instrucción después del ciclo. En la línea 68 de la figura 16.9 se decrementa `cuentaBuferOcupado` para indicar que `bufer` ahora está vacío (es decir, un consumidor no puede leer el valor, pero un productor puede colocar otro valor en `bufer`), en la línea 70 se muestra un mensaje en la ventana de consola, indicando el valor que está leyendo el consumidor y en la línea 72 se invoca al método `notify`. Si hay algún subproceso en espera del bloqueo sobre este objeto `BuferSincronizado`, uno de ellos entra al estado *Listo*, indicando que, tan pronto como se

4. La invocación al método `notify` funciona correctamente en este programa, ya que sólo un subproceso llama al método `obtener` en un momento dado (el subproceso `Consumidor`). Los programas que tienen varios subprocesos en espera de una condición deben invocar a `notifyAll` para asegurarse que todos los subprocesos reciban notificaciones en forma apropiada.

asigne un procesador al subproceso, éste podrá ahora tratar de readquirir el bloqueo y seguir realizando su tarea. El método `notify` regresa de inmediato y después el método `obtener` devuelve el valor de `bufer` al método que lo llamó.[5] Cuando el método `obtener` regresa, el bloqueo sobre el objeto `BuferSincronizado` se libera implícitamente.

La clase `PruebaBuferCompartido2` (figura 16.10) es similar a la clase `PruebaBuferCompartido` (figura 16.8). `PruebaBuferCompartido2` contiene el método `main` (líneas 6 a 28), el cual inicia la aplicación. En la línea 11 se crea una instancia de un objeto `BuferSincronizado` compartido, y se asigna su referencia a la variable `BuferSincronizado` llamada `ubicacionCompartida`. Utilizamos una variable `BuferSincronizado` en vez de una variable `Bufer`, de manera que `main` pueda invocar al método `mostrarEstado` de `BuferSincronizado`, el cual no está declarado en la interfaz `Bufer`. El objeto `BuferSincronizado` almacena los datos que se compartirán entre los subprocesos productor y consumidor. En las líneas 14 a 19 se muestran los encabezados de columna para la salida. En las líneas 22 y 23 se crea un objeto `Productor` y un objeto `Consumidor`. Cada llamada al constructor pasa el valor de `ubicacionCompartida` como argumento para el constructor, por lo que cada objeto se inicializa con una referencia al mismo objeto `Bufer`. A continuación, en las líneas 25 y 26 se invoca el método `start` en los subprocesos `productor` y `consumidor` para colocarlos en el estado *Listo*. Este método inicia a estos subprocesos y establece la llamada inicial al método `run` de cada subproceso, el cual empezará a ejecutarse cuando a cada subproceso se le asigne un procesador. Por último, el método `main` termina y el subproceso `main` de ejecución muere.

Estudie los resultados en la figura 16.10. Observe que cada entero producido se consume exactamente una vez; no se pierden valores y no se consumen valores más de una vez. La sincronización y la variable de condición aseguran que el productor y el consumidor no puedan realizar sus tareas a menos que sea su turno. El productor debe ir primero; el consumidor debe esperar si el productor no ha producido nada desde la última vez que el consumidor consumió; y el productor debe esperar si el consumidor no ha consumido todavía el valor que el productor haya producido más recientemente. Ejecute este programa varias veces para confirmar que todos los enteros producidos se consuman exactamente una vez. En el primer y segundo conjuntos de resultados, observe las líneas que indican cuando el productor y el consumidor deben esperar para realizar sus respectivas tareas. En el tercer conjunto de resultados, observe que el productor y el consumidor pudieron realizar sus tareas sin tener que esperar.

```
1  // Fig. 16.10: PruebaBuferCompartido2.java
2  // PruebaBuferCompartido2 crea los subprocesos productor y consumidor.
3
4  public class PruebaBuferCompartido2 {
5
6     public static void main( String [] args )
7     {
8        // crear objeto compartido utilizado por los subprocesos; usar una referencia
9        // BuferSincronizado en vez de una referencia Bufer, para que podamos invocar
10       // al método mostrarEstado de BuferSincronizado desde main
11       BuferSincronizado ubicacionCompartida = new BuferSincronizado();
12
13       // Mostrar encabezados de columna para los resultados
14       StringBuffer encabezadosColumna = new StringBuffer( "Operacion" );
15       encabezadosColumna.setLength( 40 );
```

Figura 16.10 Subprocesos que modifican a un objeto compartido con sincronización. (Parte 1 de 3.)

5. La invocación al método `notify` funciona correctamente en este programa, ya que sólo un subproceso llama al método `establecer` en un momento dado (el subproceso `Productor`). Los programas que tienen varios subprocesos en espera de una condición deben invocar a `notifyAll` para asegurarse que todos los subprocesos reciban notificaciones en forma apropiada.

http://librosysolucionarios.net

```
16         encabezadosColumna.append( "Bufer\t\tCuenta ocupado" );
17         System.err.println( encabezadosColumna );
18         System.err.println();
19         ubicacionCompartida.mostrarEstado( "Estado inicial" );
20
21         // crear objetos productor y consumidor
22         Productor productor = new Productor( ubicacionCompartida );
23         Consumidor consumidor = new Consumidor( ubicacionCompartida );
24
25         productor.start(); // iniciar subproceso productor
26         consumidor.start(); // iniciar subproceso consumidor
27
28      } // fin de main
29
30   } // fin de la clase PruebaBuferCompartido2
```

```
Operacion                              Bufer     Cuenta ocupado

Estado inicial                         -1        0

Consumidor trata de leer.
Bufer vacio. Consumidor espera.        -1        0

Productor escribe 1                    1         1

Consumidor lee 1                       1         0

Consumidor trata de leer.
Bufer vacio. Consumidor espera.        1         0

Productor escribe 2                    2         1

Consumidor lee 2                       2         0

Productor escribe 3                    3         1

Consumidor lee 3                       3         0

Consumidor trata de leer.
Bufer vacio. Consumidor espera.        3         0

Productor escribe 4                    4         1

Consumidor lee 4                       4         0
Productor termino de producir.
Terminando Productor.

Consumidor leyo valores, dando un total de: 10.
Terminando Consumidor.
```

Figura 16.10 Subprocesos que modifican a un objeto compartido con sincronización. (Parte 2 de 3.)


```
Operacion                              Bufer     Cuenta ocupado

Estado inicial                         -1        0

Consumidor trata de leer.
Bufer vacio. Consumidor espera.        -1        0

Productor escribe 1                    1         1

Consumidor lee 1                       1         0

Productor escribe 2                    2         1

Productor trata de escribir.
Bufer lleno. Productor espera.         2         1

Consumidor lee 2                       2         0

Productor escribe 3                    3         1

Consumidor lee 3                       3         0

Productor escribe 4                    4         1

Productor termino de producir.
Terminando Productor.
Consumidor lee 4                       4         0

Consumidor leyo valores, dando un total de: 10.
Terminando Consumidor.
```

```
Operacion                              Bufer     Cuenta ocupado

Estado inicial                         -1        0

Productor escribe 1                    1         1

Consumidor lee 1                       1         0

Productor escribe 2                    2         1

Consumidor lee 2                       2         0

Productor escribe 3                    3         1

Consumidor lee 3                       3         0

Productor escribe 4                    4         1

Productor termino de producir.
Terminando Productor.
Consumidor lee 4                       4         0

Consumidor leyo valores, dando un total de: 10.
Terminando Consumidor.
```

Figura 16.10 Subprocesos que modifican a un objeto compartido con sincronización. (Parte 3 de 3.)

http://librosysolucionarios.net

16.8 Relación productor/consumidor: Búfer circular

El programa de la sección 16.7 utiliza la sincronización de subprocesos para garantizar que dos subprocesos manipulen correctamente los datos en un búfer compartido. Sin embargo, es probable que la aplicación no tenga un rendimiento óptimo. Si los dos subprocesos operan a distintas velocidades, uno de ellos invertirá más tiempo (o la mayor parte) esperando. Por ejemplo, en el programa de la sección 16.7 compartimos una sola variable entera entre los dos subprocesos. Si el subproceso productor produce los valores con más rapidez de la que el consumidor pueda consumirlos, entonces el productor tiene que esperar al consumidor, ya que no hay más ubicaciones en memoria para colocar el siguiente valor. De manera similar, si el consumidor consume con más rapidez de la que el productor pueda producir, el consumidor tiene que esperar hasta que el productor coloque el siguiente valor en la ubicación compartida en memoria. Incluso aunque haya subprocesos que operen a las mismas velocidades relativas, durante cierto periodo estos subprocesos podrían quedar "fuera de sincronización", haciendo que uno de los subprocesos tenga que esperar al otro. No podemos hacer suposiciones acerca de las velocidades relativas de los subprocesos concurrentes asíncronos. Hay muchas interacciones que ocurren con el sistema operativo, la red, el usuario y otros componentes, las cuales pueden hacer que los subprocesos operen a distintas velocidades. Cuando esto ocurre, los subprocesos tienen que esperar. Cuando los subprocesos esperan, los programas se hacen menos eficientes, los programas que interactúan con el usuario se vuelven menos sensibles y las aplicaciones sufren de retrasos más largos, debido a que el procesador no se utiliza con eficiencia.

Para minimizar el tiempo de espera para los subprocesos que comparten recursos y operan a las mismas velocidades promedio, podemos implementar un *búfer circular* que proporcione un espacio de búfer adicional, en el cual el productor puede colocar valores y desde el cual el consumidor puede recuperar esos valores. Supongamos que el búfer se implementa como un arreglo. El productor y el consumidor trabajan desde la parte inicial del arreglo. Cuando uno de los subprocesos llega al final del arreglo, simplemente regresa al primer elemento del arreglo para realizar su siguiente tarea. Si el productor produce temporalmente valores con más rapidez de la que el consumidor pueda consumirlos, el productor puede escribir más valores en el espacio de búfer adicional (si es que hay espacio disponible). Esta capacidad permite al productor realizar su tarea, aún y cuando el consumidor no esté listo para recibir el valor actual que se está produciendo. De manera similar, si el consumidor consume con más rapidez de la que el productor pueda producir nuevos valores, el consumidor puede leer valores adicionales del búfer (si es que los hay). Esto permite al consumidor realizar su tarea, inclusive cuando el productor no esté listo para producir valores adicionales.

Observe que el búfer circular sería inapropiado si el productor y el consumidor operaran consistentemente a distintas velocidades. Si el consumidor siempre se ejecuta con más rapidez que el productor, entonces es suficiente con que el búfer contenga una ubicación. Las ubicaciones adicionales serían un desperdicio de memoria. Si el productor siempre se ejecutara con más rapidez, se requeriría un búfer con un número infinito de ubicaciones para absorber la producción adicional.

La clave para utilizar un búfer circular es proporcionarle las ubicaciones suficientes como para manejar la producción "extra" anticipada. Si, durante cierto tiempo, determinamos que el productor a menudo produce hasta tres valores más de los que el consumidor puede consumir, podemos proporcionar un búfer de cuando menos tres celdas, para manejar la producción adicional. No queremos que el búfer sea muy pequeño, ya que eso haría que los subprocesos tuvieran que esperar más. Por otro lado, no queremos que el búfer sea muy grande, ya que eso desperdiciaría memoria.

Tip de rendimiento 16.4



Aun y cuando se utilice un búfer circular, existe la posibilidad de que un subproceso productor llene el búfer, lo cual forzaría al subproceso productor a esperar hasta que el consumidor consuma un valor para liberar un elemento en el búfer. De manera similar, si el búfer está vacío en un momento dado, el subproceso consumidor deberá esperar a que el productor produzca otro valor. La clave para usar un búfer circular es optimizar el tamaño del búfer, para minimizar la cantidad de tiempo de espera en los subprocesos.

El programa de las figuras 16.11 a 16.15 demuestra cómo un productor y un consumidor utilizan un búfer circular (en este caso, un arreglo compartido de tres celdas) mediante la sincronización. En esta versión de la relación productor/consumidor, el consumidor consume un valor sólo cuando el arreglo no está vacío, y el productor produce un valor sólo cuando el arreglo no está lleno. Este programa se implementa como una aplicación

de ventana que envía su salida a un objeto `JTextArea`. Las instrucciones que crearon e iniciaron los objetos subproceso en el método `main` de la clase `PruebaBuferCompartido2` (figura 16.10) ahora aparecen en la clase `PruebaBuferCircular` (figura 16.15). Las clases `Productor` (figura 16.12) y `Consumidor` (figura 16.13) realizan las mismas tareas que en las figuras 16.5 y 16.6, respectivamente, con la excepción de que muestran mensajes en el objeto `JTextArea` de la ventana de aplicación.

Éste es el primer programa que utiliza subprocesos separados para modificar el contenido mostrado en una GUI de Swing. La naturaleza de la programación con subprocesamiento múltiple evita que el programador sepa exactamente cuándo se ejecutará un subproceso. Los componentes de swing no son *a prueba de subprocesos*; si varios subprocesos manipulan un componente de la GUI de Swing, los resultados tal vez no sean correctos. Todas las interacciones con los componentes de la GUI de Swing deben llevarse a cabo un subproceso a la vez. Generalmente, este subproceso es el *subproceso despachador de eventos* (también conocido como el *subproceso manejador de eventos*). La clase `SwingUtilities` (paquete `javax.swing`) proporciona el método estático `invokeLater` para ayudar con este proceso. El método `invokeLater` especifica que las instrucciones de procesamiento de la GUI deben ejecutarse posteriormente, como parte del subproceso despachador de eventos. El método `invokeLater` recibe como argumento un objeto que implementa a la interfaz `Runnable` (paquete `java.lang`). Toda clase que implemente a `Runnable` debe declarar un método `run` con el encabezado:

```
public void run()
```

De hecho, cada objeto `Thread` es un objeto `Runnable`, por lo que tiene un método `run`. Los subprocesos en este ejemplo, que muestran su salida en la GUI, pasan objetos de la clase `SalidaRunnable` (figura 16.11) al método `invokeLater`. Cada objeto `SalidaRunnable` recibe una referencia al objeto `JTextArea` en el que se muestra la salida, y una cadena que representa el mensaje a mostrar. El método `run` de `SalidaRunnable` (líneas 17 a 20) anexa el mensaje el objeto `JTextArea`. Cuando el programa llama a `invokeLater`, la actualización del componente la GUI se pondrá en una cola para ejecutarse en el subproceso despachador de eventos. Después, el método `run` de `SalidaRunnable` será invocado como parte del subproceso despachador de eventos para llevar a cabo la impresión de los resultados y asegurar que el componente de la GUI se actualice en forma segura para los subprocesos.

```
1  // Fig. 16.11: SalidaRunnable.java
2  // La clase SalidaRunnable actualiza el objeto JTextArea con los resultados
3  import javax.swing.*;
4
5  public class SalidaRunnable implements Runnable {
6     private JTextArea areaSalida;
7     private String mensajeParaAnexar;
8
9     // inicializar areaSalida y mensaje
10    public SalidaRunnable( JTextArea salida, String mensaje )
11    {
12       areaSalida = salida;
13       mensajeParaAnexar = mensaje;
14    }
15
16    // método llamado por SwingUtilities.invokeLater para actualizar areaSalida
17    public void run()
18    {
19       areaSalida.append( mensajeParaAnexar );
20    }
21
22 } // fin de la clase SalidaRunnable
```

Figura 16.11 Los objetos `SalidaRunnable` se pasan al método `invokeLater` de `SwingUtilities` para procesar las actualizaciones de la GUI en el subproceso despachador de eventos.

http://librosysolucionarios.net

```
1  // Fig. 16.12: Productor.java
2  // El método run de Productor controla un subproceso que
3  // almacena valores de 11 a 20 en ubicacionCompartida.
4  import javax.swing.*;
5
6  public class Productor extends Thread {
7     private Bufer ubicacionCompartida;
8     private JTextArea areaSalida;
9
10    // constructor
11    public Productor( Bufer compartido, JTextArea salida )
12    {
13       super( "Productor" );
14       ubicacionCompartida = compartido;
15       areaSalida = salida;
16    }
17
18    // almacenar valores de 11 a 20 en el búfer de ubicacionCompartida
19    public void run()
20    {
21       for ( int cuenta = 11; cuenta <= 20; cuenta++ ) {
22
23          // estar inactivo de 0 a 3 segundos, después colocar valor en Bufer
24          try {
25             Thread.sleep( ( int ) ( Math.random() * 3000 ) );
26             ubicacionCompartida.establecer( cuenta );
27          }
28
29          // si se interrumpió el subproceso inactivo, imprimir el rastreo de la pila
30          catch ( InterruptedException excepcion ) {
31             excepcion.printStackTrace();
32          }
33       }
34
35       String nombre = getName();
36       SwingUtilities.invokeLater( new SalidaRunnable( areaSalida, "\n" +
37          nombre + " termino de producir.\n" + nombre + " terminado.\n" ) );
38
39    } // fin del método run
40
41 } // fin de la clase Productor
```

Figura 16.12 La clase **Productor** representa al productor en una relación productor/consumidor.

La clase **Productor** (figura 16.12) es una versión ligeramente modificada de la que se presenta en la figura 16.5. En la línea 8 se declara un objeto **JTextArea** que será utilizado para mostrar un mensaje al final del método **run**. Esta referencia se inicializa en el constructor (líneas 11 a 16). El método **run** (líneas 19 a 39) ahora produce valores de 11 a 20. Además, en las líneas 36 y 37 se muestra un mensaje en la GUI, indicando que el subproceso terminó de producir los valores y que va a terminar su ejecución. Observe el uso de un objeto **SalidaRunnable** con el método **invokeLater** de **SwingUtilities**, para asegurar que el texto se anexe al área de texto en el subproceso despachador de eventos.

La clase **Consumidor** (figura 16.13) es una versión ligeramente modificada de la que se presenta en la figura 16.4. En la línea 8 se declara un objeto **JTextArea** para mostrar un mensaje al final del método **run**. Esta referencia se inicializa en el constructor (líneas 11 a 16). El método **run** (líneas 19 a 42) itera 10 veces. En las líneas 38 a 40 se muestra un mensaje indicando que el subproceso terminó de consumir los valores y que va

a terminar su ejecución. De nuevo, observe el uso de un objeto **SalidaRunnable** con el método **invokeLater** de **SwingUtilities** para asegurar que la GUI se actualice en el subproceso despachador de eventos.

Los cambios considerables al ejemplo de la sección 16.7 ocurren en **BuferCircular** (figura 16.14), que sustituye a **BuferSincronizado** (figura 16.9). **BuferCircular** contiene cinco campos. El arreglo **buferes** (línea 8) es un arreglo entero de tres elementos, que representa al búfer circular. La variable **cuentaBuferesOcupados** (línea 11) es la variable de condición que puede usarse para determinar si un productor puede escribir en el búfer circular (es decir, si **cuentaBuferesOcupados** es menor que el número de elementos en **buferes**) y si un consumidor puede leer del búfer circular (es decir, que **cuentaBuferesOcupados** sea mayor que 0). La variable **leerUbicacion** (línea 14) indica la posición a partir de la cual el con-

```
1  // Fig. 16.13: Consumidor.java
2  // El método run de Consumidor controla un subproceso que itera diez
3  // veces y lee un valor de ubicacionCompartida cada vez.
4  import javax.swing.*;
5
6  public class Consumidor extends Thread {
7     private Bufer ubicacionCompartida; // referencia al objeto compartido
8     private JTextArea areaSalida;
9
10    // constructor
11    public Consumidor( Bufer compartido, JTextArea salida )
12    {
13       super( "Consumidor" );
14       ubicacionCompartida = compartido;
15       areaSalida = salida;
16    }
17
18    // leer el valor de ubicacionCompartida diez veces y sumar los valores
19    public void run()
20    {
21       int suma = 0;
22
23       for ( int cuenta = 1; cuenta <= 10; cuenta++ ) {
24
25          // estar inactivo de 0 a 3 segundos, leer el valor de Bufer y sumarlo a suma
26          try {
27             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
28             suma += ubicacionCompartida.obtener();
29          }
30
31          // si se interrumpió el subproceso inactivo, imprimir el rastreo de la pila
32          catch ( InterruptedException excepcion ) {
33             excepcion.printStackTrace();
34          }
35       }
36
37       String nombre = getName();
38       SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
39          "\nTotal que consumio " + nombre + ": " + suma + "\n" +
40          nombre + " terminado.\n" ) );
41
42    } // fin del método run
43
44 } // fin de la clase Consumidor
```

Figura 16.13 La clase **Consumidor** representa al consumidor en una relación productor/consumidor.

http://librosysolucionarios.net

sumidor puede leer el siguiente valor. La variable **escribirUbicacion** (línea 14) indica la siguiente ubicación en la que el productor puede colocar un valor. Todos los resultados se muestran en **areaSalida** (línea 17).

El método **establecer** de **BuferCircular** (líneas 26 a 68) realiza las mismas tareas que en la figura 16.9, con unas cuantas modificaciones. El ciclo **while** en las líneas 32 a 47 determina si el productor debe

```
1  // Fig. 16.14: BuferCircular.java
2  // BuferCircular sincroniza el acceso a un arreglo de búferes compartidos.
3  import javax.swing.*;
4
5  public class BuferCircular implements Bufer {
6
7     // cada elemento del arreglo es un bufer
8     private int buferes[] = { -1, -1, -1 };
9
10    // cuentaBuferesOcupados mantiene la cuenta de búferes ocupados
11    private int cuentaBuferesOcupados = 0;
12
13    // variables que mantienen las ubicaciones de lectura y escritura en el búfer
14    private int ubicacionLectura = 0, ubicacionEscritura = 0;
15
16    // referencia al componente de la GUI que muestra la salida
17    private JTextArea areaSalida;
18
19    // constructor
20    public BuferCircular( JTextArea salida )
21    {
22       areaSalida = salida;
23    }
24
25    // colocar valor en bufer
26    public synchronized void establecer( int valor )
27    {
28       // obtener nombre del subproceso que llamó a este método, para mostrarlo
            en pantalla
29       String nombre = Thread.currentThread().getName();
30
31       // mientras no haya ubicaciones vacías, colocar subproceso en estado de espera
32       while ( cuentaBuferesOcupados == buferes.length ) {
33
34          // mostrar información de subproceso y de bufer, después esperar
35          try {
36             SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
37                "\nTodos los búferes llenos. " + nombre + " espera." ) );
38             wait();
39          }
40
41          // si se interrumpió el proceso en espera, imprimir el rastreo de la pila
42          catch ( InterruptedException excepcion )
43          {
44             excepcion.printStackTrace();
45          }
46
47       } // fin de instrucción while
48
```

Figura 16.14 La clase **BuferCircular** supervisa el acceso a un arreglo de enteros compartido. (Parte 1 de 3.)


```
49       // colocar valor en ubicacionEscritura de búferes
50       buferes[ ubicacionEscritura ] = valor;
51
52       // actualizar componente de la GUI de Swing con el valor producido
53       SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
54          "\n" + nombre + " escribe " + buferes[ ubicacionEscritura ] ) );
55
56       // acaba de producir un valor, por lo que se incrementa el número de búferes ocupados
57       ++cuentaBuferesOcupados;
58
59       // actualizar ubicacionEscritura para la siguiente operación de escritura
60       ubicacionEscritura = ( ubicacionEscritura + 1 ) % buferes.length;
61
62       // mostrar contenido de búferes compartidos
63       SwingUtilities.invokeLater( new SalidaRunnable(
64          areaSalida, creaSalidaEstado() ) );
65
66       notify(); // regresar el subproceso en espera (si hay uno) al estado listo
67
68    } // fin del método establecer
69
70    // devolver valor de bufer
71    public synchronized int obtener()
72    {
73       // obtener nombre del subproceso que llamó a este método, para mostrarlo
            en pantalla
74       String nombre = Thread.currentThread().getName();
75
76       // mientras no haya datos que leer, colocar el subproceso en estado de espera
77       while ( cuentaBuferesOcupados == 0 ) {
78
79          // mostrar información de subproceso y de bufer, después esperar
80          try {
81             SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
82                "\nTodos los búferes vacíos. " + nombre + " espera." ) );
83             wait();
84          }
85
86          // si se interrumpió el subproceso en espera, imprimir el rastreo de la pila
87          catch ( InterruptedException excepcion ) {
88             excepcion.printStackTrace();
89          }
90
91       } // fin de instrucción while
92
93       // obtener valor en ubicacionLectura actual
94       int valorLectura = buferes[ ubicacionLectura ];
95
96       // actualizar componente de la GUI de Swing con el valor consumido
97       SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
98          "\n" + nombre + " lee " + valorLectura + " " ) );
99
100      // acaba de consumir un valor, por lo que se decrementa el número de búferes ocupados
101      --cuentaBuferesOcupados;
102
```

Figura 16.14 La clase **BuferCircular** supervisa el acceso a un arreglo de enteros compartido. (Parte 2 de 3.)

http://libros...

```
103        // actualizar ubicacionLectura para la siguiente operación de lectura
104        ubicacionLectura = ( ubicacionLectura + 1 ) % buferes.length;
105
106        // mostrar contenido de búferes compartidos
107        SwingUtilities.invokeLater( new SalidaRunnable(
108           areaSalida, crearSalidaEstado() ) );
109
110        notify(); // regresar el subproceso en espera (si hay uno) al estado listo
111
112        return valorLectura;
113
114     } // fin del método obtener
115
116     // crear salida de estado
117     public String crearSalidaEstado()
118     {
119        // primera línea de información de estado
120        String salida =
121           "(búferes ocupados: " + cuentaBuferesOcupados + ")\nbúferes: ";
122
123        for ( int i = 0; i < buferes.length; i++ )
124           salida += " " + buferes[ i ] + "  ";
125
126        // segunda línea de información de estado
127        salida += "\n          ";
128
129        for ( int i = 0; i < buferes.length; i++ )
130           salida += "---- ";
131
132        // tercera línea de información de estado
133        salida += "\n          ";
134
135        // anexar indicadores de ubicacionLectura (L) y ubicacionEscritura (E)
136        // debajo de las ubicaciones de búfer apropiadas
137        for ( int i = 0; i < buferes.length; i++ )
138
139           if ( i == ubicacionEscritura && ubicacionEscritura == ubicacionLectura )
140              salida += " EL  ";
141           else if ( i == ubicacionEscritura )
142              salida += " E   ";
143           else if ( i == ubicacionLectura )
144              salida += "  L  ";
145           else
146              salida += "     ";
147
148        salida += "\n";
149
150        return salida;
151
152     } // fin del método crearSalidaEstado
153
154  } // fin de la clase BuferCircular
```

Figura 16.14 La clase **BuferCircular** supervisa el acceso a un arreglo de enteros compartido. (Parte 3 de 3.)

http://librosysolucionarios.net

esperar (es decir, que todos los búferes estén llenos). De ser así, en las líneas 36 y 37 se invoca al método **invokeLater** de **SwingUtilities** para actualizar la salida e indicar que el consumidor está esperando realizar su tarea. Después en la línea 38 se invoca al método **wait** para colocar el subproceso productor en el estado *En espera* por el objeto **BuferCircular**. Cuando la ejecución continúa de un modo u otro en la línea 50, después del ciclo **while**, el valor escrito por el productor se coloca en el búfer circular, en la ubicación **ubicacionEscritura**. A continuación, en las líneas 53 y 54 se invoca al método **invokeLater** de **SwingUtilities** para actualizar la salida en la GUI con el valor producido. En la línea 57 se incrementa **cuentaBuferesOcupados**, ya que ahora hay por lo menos un valor en el búfer que el consumidor puede leer. Después, en la línea 60 se actualiza **ubicacionEscritura** para la siguiente llamada al método **establecer** de **BuferCircular**. La salida continúa en las líneas 63 y 64, invocando al método **crearEstadoSalida** (declarado en las líneas 117 a 152), el cual muestra el número de búferes ocupados, el contenido de los búferes y los valores actuales de **ubicacionEscritura** y **ubicacionLectura**. Finalmente, en la línea 66 se invoca al método **notify** para indicar que un subproceso que se encuentre en espera por el objeto **BuferSincronizado** (si es que hay uno en espera) debe cambiar al estado *Listo*.

El método **obtener** (líneas 71 a 114) de la clase **BuferCircular** también realiza las mismas tareas que en la figura 16.9, con unas cuantas modificaciones. El ciclo **while** en las líneas 77 a 91 determina si el consumidor debe esperar (es decir, que todos los búferes estén vacíos). Si el subproceso consumidor debe esperar, en las líneas 81 y 82 se invoca al método **invokeLater** de **SwingUtilities** para actualizar la salida e indicar que el consumidor está esperando realizar su tarea. Después, en la línea 83 se invoca al método **wait** para colocar el subproceso consumidor en el estado *En espera* por el objeto **BuferCircular**. Cuando la ejecución continúa de un modo u otro en la línea 94, después del ciclo **while**, a **valorLectura** se le asigna el valor de la ubicación **ubicacionLectura** en el búfer circular. En las líneas 97 y 98 se invoca al método **invokeLater** de **SwingUtilities** para actualizar la salida en la GUI con el valor consumido. En la línea 101 se decrementa **cuentaBuferesOcupados**, ya que hay por lo menos una posición libre en el búfer, en donde el subproceso productor puede colocar un valor. Después, en la línea 104 se actualiza **ubicacionLectura** para la siguiente llamada al método **obtener** de **BuferCircular**. En las líneas 107 y 108 se invoca al método **crearEstadoSalida** para mostrar el número de búferes ocupados, el contenido de los búferes y los valores actuales de **ubicacionEscritura** y **ubicacionLectura**. Finalmente, en la línea 110 se invoca al método **notify** para hacer que el siguiente subproceso que espera el objeto **BuferCircular** cambie al estado *Listo*, y en la línea 112 se devuelve el valor consumido al método que hizo la llamada.

La clase **PruebaBuferCircular** (figura 16.15) contiene el método **main** que inicia la aplicación. En la línea 43 se crea un objeto **PruebaBuferCircular**. El constructor crea la GUI de la aplicación (líneas

```
1  // Fig. 16.15: PruebaBuferCircular.java
2  // PruebaBuferCircular muestra a dos subprocesos manipulando un búfer circular.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  // establecer los subprocesos productor y consumidor e iniciarlos
8  public class PruebaBuferCircular extends JFrame {
9     JTextArea areaSalida;
10
11    // configurar la GUI
12    public PruebaBuferCircular()
13    {
14       super( "Demostración de sincronización de subprocesos" );
15
16       areaSalida = new JTextArea( 20,30 );
17       areaSalida.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
```

Figura 16.15 La clase **PruebaBuferCircular** configura una aplicación de GUI tipo productor/consumidor y crea instancias de los subprocesos productor y consumidor. (Parte 1 de 4.)

```
18          getContentPane().add( new JScrollPane( areaSalida ) );
19
20          setSize( 310, 500 );
21          setVisible( true );
22
23          // crear objeto compartido utilizado por los subprocesos; utilizamos una referencia
24          // BuferCircular en vez de una referencia Bufer, para poder invocar al
25          // método crearSalidaEstado de BuferCircular
26          BuferCircular ubicacionCompartida = new BuferCircular( areaSalida );
27
28          // mostrar el estado inicial de los búferes en BuferCircular
29          SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
30             ubicacionCompartida.crearSalidaEstado() ) );
31
32          // establecer subprocesos
33          Productor productor = new Productor( ubicacionCompartida, areaSalida );
34          Consumidor consumidor = new Consumidor( ubicacionCompartida, areaSalida );
35
36          productor.start(); // iniciar subproceso productor
37          consumidor.start(); // iniciar subproceso consumidor
38
39       } // fin del constructor
40
41       public static void main ( String args[] )
42       {
43          PruebaBuferCircular aplicacion = new PruebaBuferCircular();
44          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
45       }
46
47    } // fin de la clase PruebaBuferCircular
```

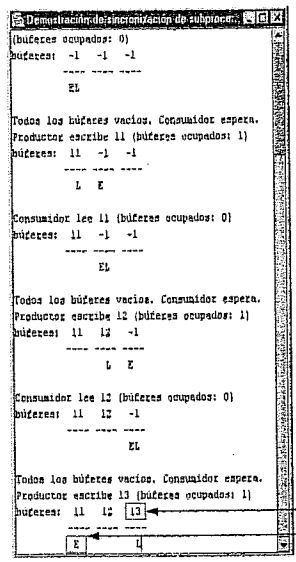
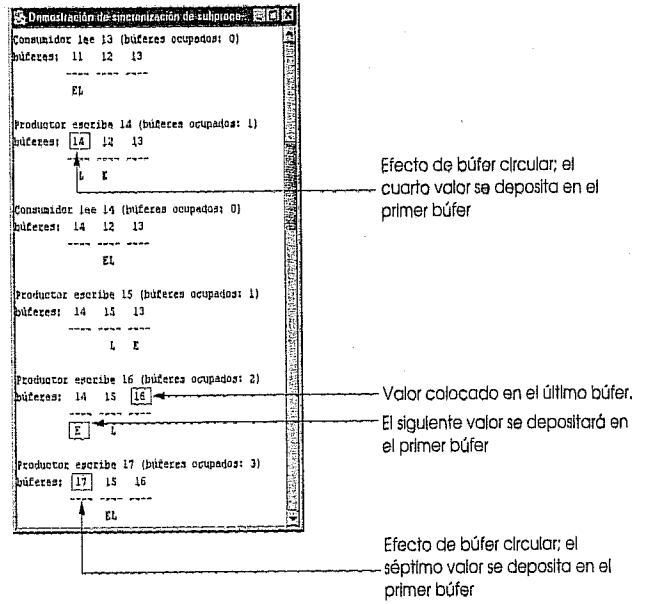


Figura 16.15 La clase **PruebaBuferCircular** configura una aplicación de GUI tipo productor/consumidor y crea instancias de los subprocesos productor y consumidor. (Parte 2 de 4.)



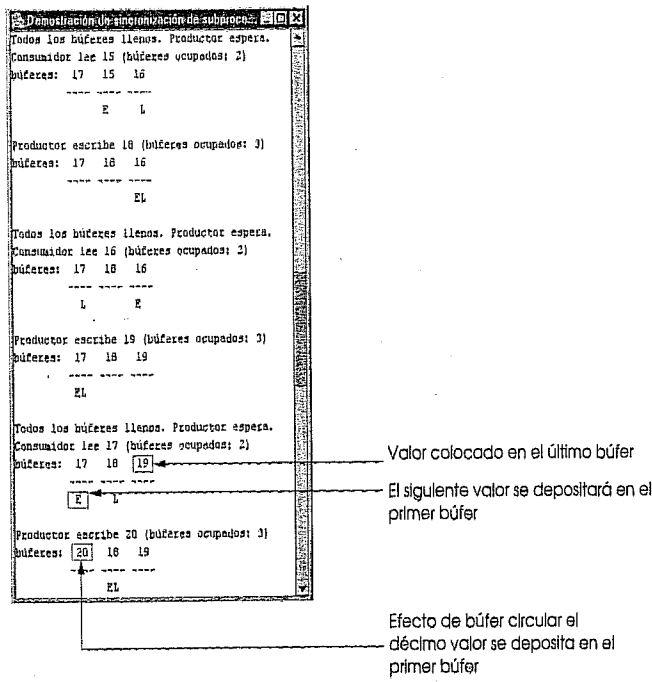


Figura 16.15 La clase **PruebaBuferCircular** configura una aplicación de GUI tipo productor/consumidor y crea instancias de los subprocesos productor y consumidor. (Parte 3 de 4.)

http://librosysolucionarios.net

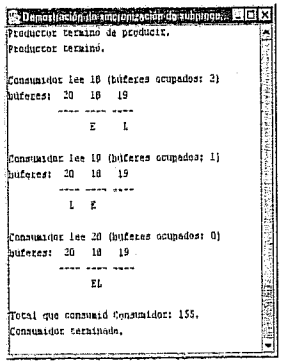


Figura 16.15 La clase `PruebaBuferCircular` configura una aplicación de GUI tipo productor/consumidor y crea instancias de los subprocesos productor y consumidor. (Parte 4 de 4.)

16 a 21), crea el objeto `Bufer` de la clase `BuferCircular` (línea 26) y crea los subprocesos (líneas 33 y 34); después los inicia (líneas 36 y 37). En los resultados se incluyen el valor actual de `cuentaBuferesOcupados`, el contenido de los búferes y los valores actuales de `ubicacionEscritura` y `ubicacionLectura`. En los resultados, las letras `E` y `L` representan los valores actuales de `ubicacionEscritura` y `ubicacionLectura`, respectivamente. Observe que, después de colocar el tercer valor en el tercer elemento del búfer, el cuarto valor se inserta al principio del arreglo. Esto proporciona un efecto de búfer circular. El método `crearEstadoSalida` de la clase `BuferSincronizado` (figura 16.14, líneas 117 a 152) da formato al contenido del arreglo `buferes`.

16.9 Subprocesos tipo "daemon"

Un subproceso daemon es un subproceso que se ejecuta para beneficio de otros subprocesos. A diferencia de los subprocesos de usuario convencionales (es decir, cualquier subproceso, que no sea daemon, en un programa), los subprocesos daemon no evitan que un programa termine. Algunas implementaciones del mecanismo de recolección de basura de Java utilizan subprocesos daemon. Los subprocesos que no son daemon son subprocesos de usuario convencionales o subprocesos como el subproceso despachador de eventos, para procesar eventos de la GUI. Podemos designar un subproceso como daemon, llamando a su método `setDaemon` con un argumento `true`. Un programa puede incluir una mezcla de subprocesos daemon y subprocesos que no son daemon. Sólo cuando quedan subprocesos daemon en un programa, éste termina. Si un subproceso va a ser daemon, debe establecerse como tal antes de que su método `start` sea llamado, o antes de que se lance una excepción `IllegalThreadStateException`. El método `isDaemon` devuelve `true` si un subproceso es daemon y `false` en caso contrario.



Error común de programación 16.4

Al tratar de hacer que un subproceso sea daemon una vez que haya empezado, se produce una excepción `IllegalThreadStateException`.



Observación de ingeniería de software 16.4

El subproceso despachador de eventos es un ciclo infinito y no es un subproceso daemon. Como tal, el subproceso despachador de eventos no terminará en una aplicación de ventana, sino hasta que la aplicación llame al método `exit` de `System`.



Buena práctica de programación 16.1

No asigne tareas críticas a un subproceso daemon. Estas tareas se terminarán sin advertencia, lo que evitará que se completen correctamente.

16.10 Interfaz `Runnable`

Hasta ahora hemos extendido la clase `Thread` para crear nuevas clases que soporten el subprocesamiento múltiple. Sobrescribimos el método `run` para especificar las tareas a realizar concurrentemente. No obstante, si deseamos soportar el subprocesamiento múltiple en una clase que ya extiende a otra clase distinta de `Thread`, debemos implementar la interfaz `Runnable` en esa clase, ya que Java no permite que una clase extienda a más de una clase a la vez. Como lo mencionamos en la sección 16.7, la clase `Thread` implementa a la interfaz `Runnable`, según lo expresado en el encabezado de la clase:

```
public class Thread extends Object implements Runnable
```

Al implementar la interfaz `Runnable` en una clase, un programa puede manipular objetos de esa clase como objetos `Runnable`. Como en el caso cuando se extiende a la clase `Thread`, el código que ejecuta el subproceso se coloca en el método `run`.

Un programa que utiliza un objeto `Runnable` para controlar un subproceso crea un objeto `Thread` y asocia el objeto `Runnable` con ese objeto `Thread`. La clase `Thread` proporciona cinco constructores que pueden recibir referencias a objetos `Runnable` como argumentos. Por ejemplo, el constructor:

```
public Thread( Runnable objetoRunnable )
```

especifica que el método `run` de `objetoRunnable` es el método que deberá invocarse cuando el subproceso empiece a ejecutarse. El constructor:

```
public Thread( Runnable objetoRunnable, String nombreSubproceso )
```

crea un objeto `Thread` con el nombre `nombreSubproceso` y especifica que el método `run` de su argumento `objetoRunnable` es el método que deberá invocarse cuando el subproceso empiece a ejecutarse. Como siempre, debe hacerse una llamada al método `start` del objeto del subproceso para poder empezar la ejecución del subproceso.



Observación de ingeniería de software 16.5

Es más conveniente implementar la interfaz `Runnable` que extender a la clase `Thread`. Los objetos `Runnable` pueden usarse en muchas situaciones en las que los objetos `Thread` serían incómodos.

En la figura 16.16 se demuestra el uso de un subprograma con dos clases internas, en donde cada una de ellas implementa a la interfaz `Runnable`: una controla los subprocesos creados en el subprograma y la otra se utiliza con el método `invokeLater` de `SwingUtilities` para asegurar que la GUI se actualice de manera correcta. En este ejemplo también demostramos cómo suspender un subproceso (es decir, cómo evitar que se ejecute temporalmente), cómo reanudar un subproceso suspendido y cómo terminar un subproceso en ejecución hasta que una condición se vuelva falsa. (*Nota:* La clase `Thread` ya cuenta con los métodos `suspend`, `resume` y `stop`; sin embargo, estos métodos están *desaprobados* y no deberán utilizarse más en los programas de Java. Los programas que utilizan estos métodos podrían encontrarse con problemas como el interbloqueo con `suspend` y la corrupción de datos con `stop`, ya que los métodos `suspend`, `resume` y `stop` no funcionan de manera apropiada con los monitores. Por estas razones demostraremos técnicas de codificación para suspender, reanudar y detener subprocesos. Como veremos, estos mecanismos se basan en instrucciones sincronizadas, ciclos y variables de bandera `boolean`.)

```
1  // Fig. 16.16: CaracteresAleatorios.java
2  // La clase CaracteresAleatorios demuestra el uso de la interfaz Runnable
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class CaracteresAleatorios extends JApplet implements ActionListener {
8     private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
9     private final static int TAMANIO = 3;
```

Figura 16.16 Interfaz `Runnable`, suspendiendo y reanudando subprocesos. (Parte 1 de 4.)

http://librosysolucionarios.net

```
10    private JLabel salidas[];
11    private JCheckBox casillasVerificacion[];
12    private Thread subprocesos[];
13    private boolean suspendido[];
14
15    // configurar la GUI y establecer arreglos
16    public void init()
17    {
18       salidas = new JLabel[ TAMANIO ];
19       casillasVerificacion = new JCheckBox[ TAMANIO ];
20       subprocesos = new Thread[ TAMANIO ];
21       suspendido = new boolean[ TAMANIO ];
22
23       Container contenedor = getContentPane();
24       contenedor.setLayout( new GridLayout( TAMANIO, 2, 5, 5 ) );
25
26       // crear componentes de la GUI, registrar componentes de escucha y adjuntar
27       // componentes al panel de contenido
28       for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ ) {
29          salidas[ cuenta ] = new JLabel();
30          salidas[ cuenta ].setBackground( Color.GREEN );
31          salidas[ cuenta ].setOpaque( true );
32          contenedor.add( salidas[ cuenta ] );
33
34          casillasVerificacion[ cuenta ] = new JCheckBox( "Suspendido" );
35          casillasVerificacion[ cuenta ].addActionListener( this );
36          contenedor.add( casillasVerificacion[ cuenta ] );
37       }
38
39    } // fin del método init
40
41    // crear e iniciar subprocesos cada vez que se hace una llamada a start (es decir, después de
42    // init y cuando el usuario vuelve a visitar la página Web que contiene a este subprograma)
43    public void start()
44    {
45       for ( int cuenta = 0; cuenta < subprocesos.length; cuenta++ ) {
46
47          // crear objeto Thread; inicializar objeto que implementa a Runnable
48          subprocesos[ cuenta ] =
49             new Thread( new ObjetoRunnable(), "Subproceso" + ( cuenta + 1 ) );
50
51          subprocesos[ cuenta ].start(); // empezar la ejecución del objeto Thread
52       }
53    }
54
55    // determinar ubicación del subproceso en el arreglo subprocesos
56    private int obtenerIndice( Thread actual )
57    {
58       for ( int cuenta = 0; cuenta < subprocesos.length; cuenta++ )
59          if ( actual == subprocesos[ cuenta ] )
60             return cuenta;
61
62       return -1;
63    }
64
```

Figura 16.16 Interfaz `Runnable`, suspendiendo y reanudando subprocesos. (Parte 2 de 4.)


```
65    // este método se llama cuando el usuario cambia de páginas Web; detiene todos los subprocesos
66    public synchronized void stop()
67    {
68       // establecer referencias en null para terminar el método run de cada subproceso
69       for ( int cuenta = 0; cuenta < subprocesos.length; cuenta++ )
70          subprocesos[ cuenta ] = null;
71
72       notifyAll(); // notificar a todos los subprocesos en espera para que puedan terminar
73    }
74
75    // manejar eventos de botón
76    public synchronized void actionPerformed( ActionEvent evento )
77    {
78       for ( int cuenta = 0; cuenta < casillasVerificacion.length; cuenta++ ) {
79
80          if ( evento.getSource() == casillasVerificacion[ cuenta ] ) {
81             suspendido[ cuenta ] = !suspendido[ cuenta ];
82
83             // cambiar color de la etiqueta al suspender/reanudar
84             salidas[ cuenta ].setBackground(
85                suspendido[ cuenta ] ? Color.RED : Color.GREEN );
86
87             // si el subproceso se reanudó, asegúrese de que empiece a ejecutarse
88             if ( !suspendido[ cuenta ] )
89                notifyAll();
90
91             return;
92          }
93       }
94
95    } // fin del método actionPerformed
96
97    // clase interna privada que implementa a Runnable para controlar los subprocesos
98    private class ObjetoRunnable implements Runnable {
99
100      // colocar caracteres aleatorios en la GUI, las variables subprocesoActual e
101      // índice son finales, para poder usarlas en una clase interna anónima
102      public void run()
103      {
104         // obtener referencia al subproceso en ejecución
105         final Thread subprocesoActual = Thread.currentThread();
106
107         // determinar la posición del subproceso en el arreglo
108         final int indice = obtenerIndice( subprocesoActual );
109
110         // la condición del ciclo determina cuándo debe detenerse el subproceso; el ciclo
111         // termina cuando la referencia subprocesos[ índice ] se vuelve null
112         while ( subprocesos[ indice ] == subprocesoActual ) {
113
114            // estar inactivo de 0 a 1 segundo
115            try {
116               Thread.sleep( ( int ) ( Math.random() * 1000 ) );
117
118               // determinar si el subproceso debe suspender su ejecución;
119               // sincronizar con el objeto de subprograma CaracteresAleatorios
```

Figura 16.16 Interfaz `Runnable`, suspendiendo y reanudando subprocesos. (Parte 3 de 4.)

http://librosysolucionarios.net

```
120                synchronized( CaracteresAleatorios.this ) {
121
122                   while ( suspendido[ indice ] &&
123                      subprocesos[ indice ] == subprocesoActual ) {
124
125                      // suspender temporalmente la ejecución del subproceso
126                      CaracteresAleatorios.this.wait();
127                   }
128                } // fin del bloque synchronized
129
130             } // fin de bloque try
131
132             // si el subproceso se interrumpió durante su
                   espera/inactividad, imprimir el rastreo de la pila
133             catch ( InterruptedException excepcion ) {
134                excepcion.printStackTrace();
135             }
136
137             // mostrar carácter en objeto JLabel correspondiente
138             SwingUtilities.invokeLater(
139                new Runnable() {
140
141                   // elegir caracter aleatorio y mostrarlo
142                   public void run()
143                   {
144                      char mostrarChar =
145                         alfabeto.charAt( ( int ) ( Math.random() * 26 ) );
146
147                      salidas[ indice ].setText(
148                         subprocesoActual.getName() + ": " + mostrarChar );
149                   }
150
151                } // fin de la clase interna
152
153             ); // fin de la llamada a SwingUtilities.invokeLater
154
155          } // fin de instrucción while
156
157          System.err.println( subprocesoActual.getName() + " terminando" );
158
159       } // fin del método run
160
161    } // fin de la clase interna privada ObjetoRunnable
162
163 } // fin de la clase CaracteresAleatorios
```

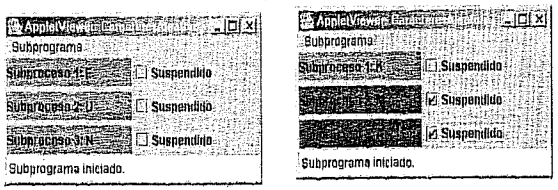


Figura 16.16 Interfaz `Runnable`, suspendiendo y reanudando subprocesos. (Parte 4 de 4.)

http://librosysolucionarios.net

La clase de subprograma `CaracteresAleatorios` muestra tres objetos `JLabel` y tres objetos `JCheckBox`. Un subproceso separado de ejecución se asocia con cada par de objetos `JLabel` y `JCheckBox`. Cada subproceso muestra letras del alfabeto en forma aleatoria, en su correspondiente objeto `JLabel`. El subprograma declara la cadena `alfabeto` (línea 8), la cual contiene las letras de la A a la Z. Esta cadena se comparte entre los tres subprocesos. Por lo general, cuando se utilizan subprocesos en un subprograma, el método `start` del subprograma crea los subprocesos y los coloca en el estado *Listo*, y el método `stop` del subprograma termina los subprocesos. El método `start` de este subprograma (líneas 43 a 53) crea una instancia de tres objetos `Thread` (líneas 48 y 49) e inicializa cada uno con una instancia de la clase `ObjetoRunnable` (declarado en las líneas 98 a 161), la cual implementa a la interfaz `Runnable`. En la línea 51 se invoca al método `start` de `Thread`, para colocar a los subprocesos en el estado *Listo*.

La clase `ObjetoRunnable` (líneas 98 a 161) implementa al método `run` de `Runnable` (líneas 102 a 159). Este método declara dos variables locales. En la línea 105 se utiliza el método `currentThread` de `Thread` para determinar cuál objeto `Thread` se está ejecutando en ese momento. En la línea 108 se hace una llamada al método utilitario del subprograma llamado `obtenerIndice` (declarado en las líneas 56 a 63), para determinar el índice del subproceso (que se está ejecutando en ese momento) en el arreglo `subprocesos`, el cual contiene referencias a los subprocesos en el subprograma. El subproceso actual muestra un carácter aleatorio en el objeto `JLabel` que tiene el mismo índice en el arreglo `salidas` (el arreglo de objetos `JLabel`). El ciclo en las líneas 112 a 155 continúa ejecutándose, siempre y cuando `subindices[ indice ]` haga referencia al subproceso que se encuentra actualmente en ejecución (`subprocesoActual`). Como veremos, `subprocesos[ indice ]` se vuelve `null` cuando es tiempo de que termine el subproceso. En ese punto, `subprocesos[ indice ]` y `subprocesoActual` ya no hacen referencia al mismo objeto, por lo que el ciclo termina y el subproceso muere. En cada iteración del ciclo, el subproceso queda inactivo durante un intervalo aleatorio de 0 a 1 segundo (línea 116).

Cuando el usuario hace clic en el objeto `JCheckBox` a la derecha de un objeto `JLabel` específico, el correspondiente objeto `Thread` deberá *suspenderse* (evitar temporalmente que se ejecute) o *reanudarse* (permitirle que siga ejecutándose). Los procesos de suspender y reanudar un subproceso pueden implementarse mediante el uso de la sincronización y de los métodos `wait` y `notify` de `Object`. En las líneas 120 a 128 se declara una instrucción sincronizada para suspender el objeto `Thread` que se esté ejecutando en esos momentos. Cuando el objeto `Thread` llega a la instrucción sincronizada, el objeto del subprograma se bloquea y la condición del ciclo (líneas 122 y 123) evalúa a `suspendido[ indice ]`, para determinar si el objeto `Thread` debe suspenderse. De ser así, y si el subproceso no ha terminado, en la línea 126 se invoca al método `wait` en el objeto del subprograma, para colocar al objeto `Thread` en el estado *En espera* por el bloqueo del subprograma. Cuando el objeto `Thread` debe reanudarse, el programa indica a todos los subprocesos en espera que estén listos para ejecutarse (de lo que hablaremos en breve). No obstante, `suspendido[ indice ]` será falso solamente para el subproceso reanudado, por lo que sólo ese subproceso tendrá la oportunidad de ejecutarse. Los demás subprocesos suspendidos (si los hay) volverán a entrar al estado *En espera*, debido al ciclo de las líneas 122 a 127. En las líneas 138 a 153 se utiliza el método `invokeLater` de `SwingUtilities` para actualizar el objeto `JLabel` del subproceso apropiado. En este ejemplo se utiliza una clase interna (líneas 139 a 151) para implementar la interfaz `Runnable`, y se pasa el objeto de la clase interna a `invokeLater`. En las líneas 144 y 145 se selecciona un carácter aleatorio de `alfabeto`. En las líneas 147 y 148 se muestra el carácter en el objeto `JLabel` apropiado. En las líneas 120 y 126, observe el uso de `CaracteresAleatorios.this` para acceder a la referencia `this` de la clase de subprograma desde la clase interna.

Si el usuario hace clic en la casilla de verificación **Suspendido** a un lado de un objeto `JLabel` específico, el programa invoca al método `actionPerformed` (líneas 76 a 95) para determinar cuál fue la casilla de verificación que generó el evento. Utilizando el índice de esa casilla de verificación en el arreglo `salidas`, en la línea 81 se cambia el valor `boolean` correspondiente en el arreglo `suspendido`. En las líneas 84 y 85 se establece el color de fondo del objeto `JLabel` a rojo si el subproceso se va a suspender, y a verde si el subproceso se va a reanudar. Si la variable `boolean` apropiada es `false`, el programa llama al método `notifyAll` de `Object` (línea 89) en el subprograma. Esto pasa a todos los subprocesos en espera del subprograma al estado *Listo*, y los prepara para reanudar su ejecución. Cuando se despacha cada subproceso al procesador para reanudar su ejecución, la condición `while` de las líneas 122 y 123 en el método `run` de `ObjetoRunnable` falla para el subproceso reanudado y el ciclo termina, lo cual permite al subproceso seguir ejecutándose desde la línea 138. Para cualquier otro subproceso que haya pasado al estado *Listo*, pero que aún esté

suspendido, la condición de las líneas 122 y 123 permanecerá siendo verdadera y el subproceso regresará al estado *En espera*.

Recuerde que en la sección 6.10 vimos que el método `stop` del subprograma se invoca cuando el usuario visita otra página Web. Este subprograma sobrescribe el método `stop` (línea 66 a 73) para terminar los tres subprocesos si el usuario sale de la página Web en la que reside este subprograma.

Tip para prevenir errores 16.4

*Puede simular la ejecución del método `stop` del subprograma en el `appletviewer`, seleccionando la opción **Parar** del menú **Subprograma**.*

El ciclo `for` de las líneas 69 y 70 establece en `null` cada una de las referencias `Thread` en el arreglo `subprocesos`. En la línea 72 se invoca el método `notifyAll` de `Object` para asegurar que todos los subprocesos en espera estén listos para ejecutarse. Cualquier subproceso que haya sido suspendido anteriormente, saldrá del ciclo en las líneas 122 a 127 y completará su iteración del ciclo `while` que empieza en la línea 112. Cuando cada subproceso encuentra la condición del ciclo `while` en la línea 112, la condición falla y el método `run` termina. Por lo tanto, cada uno de los subprocesos muere. Si el usuario regresa a la página Web, el contenedor de subprogramas llama al método `start` del subprograma para instanciar y empezar tres nuevos subprocesos.

Tip de rendimiento 16.5

Detener los subprocesos de un subprograma al salir de una página Web es una práctica de programación cortés, ya que evita que su subprograma utilice tiempo del procesador (lo cual puede reducir el rendimiento) en el equipo del navegador cuando el subprograma no se esté viendo. Los subprocesos pueden reiniciarse desde el método `start` del subprograma, el cual es invocado por el navegador cuando el usuario vuelve a visitar la página Web.

Tip para prevenir errores 16.5

*Puede simular la ejecución del método `start` del subprograma en el `appletviewer`, seleccionando la opción **Iniciar** del menú **Subprograma**.*

En este capítulo presentamos las herramientas de subprocesamiento integradas en Java, y demostramos la poderosa técnica de la sincronización de subprocesos. Además, demostramos cómo pueden usarse las técnicas de sincronización para suspender, reanudar y detener subprocesos de ejecución. El subprocesamiento múltiple es un tema avanzado, que podría cubrir todo un libro. Para obtener más ejemplos del subprocesamiento múltiple en Java, visite:

```
developer.java.sun.com/developer/technicalArticles/Threads
```

Hay numerosos artículos que hablan sobre el subprocesamiento múltiple en el contexto de otras tecnologías de Java, en:

```
developer.java.sun.com/developer/technicalArticles
```

Además, en los archivos de la revista Java en línea *JavaWorld* (`www.javaworld.com`) se proporcionan muchos artículos sobre el subprocesamiento múltiple en Java. Utilizaremos el subprocesamiento múltiple de nuevo en el capítulo 18, para ayudar a crear servidores con subprocesamiento múltiple que pueden interactuar con varios clientes concurrentemente.

16.11 (Ejemplo práctico opcional) Acerca de los objetos: Subprocesamiento múltiple

Los objetos reales llevan a cabo sus operaciones de manera independiente unos de otros, y de manera concurrente (en paralelo). Como aprendió en este capítulo, Java es un lenguaje de programación con subprocesamiento múltiple que facilita la implementación de actividades concurrentes. UML también soporta el modelado de sistemas concurrentes, como veremos en breve. En esta sección veremos cómo se beneficia nuestra simulación del subprocesamiento múltiple.

En la sección 11.9, encontramos un problema con el diagrama de colaboración de la figura 11.28: el objeto `pasajeroEsperando` (la `Persona` que espera viajar en el `Elevador`) entra incorrectamente al

http://librosysolucionarios.net

`Elevador` antes de que el objeto `pasajeroViajando` (la `Persona` que viaja en el `Elevador`) salga. El uso apropiado del subprocesamiento múltiple en Java evita este problema, al garantizar que el `pasajeroEsperando` esperará a que el `pasajeroViajando` salga del `Elevador`; como pasaría en la vida real. Utilizaremos los mecanismos de sincronización de subprocesos en Java para implementar este comportamiento cortés, de manera que se garantice que sólo una `Persona` podrá ocupar el `Elevador` en un momento dado, y para asegurar que las puertas no lastimen a los objetos `Persona` al cerrar antes de tiempo.

Subprocesos, clases activas y secuencias de modelado

Java utiliza subprocesos (flujos de control del programa independientes de otros flujos) para ejecutar actividades independientes en forma concurrente. UML proporciona la noción de una *clase activa* para representar a un subproceso o proceso. Las clases `Elevador` y `Persona` son clases activas, porque sus objetos deben ser capaces de operar concurrentemente y en forma independiente unos de otros, y de los demás objetos en el sistema. Por ejemplo, el `Elevador` debe ser capaz de moverse entre los objetos `Piso` mientras una `Persona` camina sobre un `Piso`.

Para modelar esto y para modelar las interacciones entre los objetos, presentamos ahora el *diagrama de secuencia*. Al igual que el diagrama de colaboración, el diagrama de secuencia muestra las interacciones entre objetos; sin embargo, el diagrama de secuencia enfatiza la manera en que se envían los mensajes entre objetos *a través del tiempo*. Ambos diagramas modelan las interacciones en un sistema. Los diagramas de colaboración enfatizan cuáles objetos interactúan en un sistema, y los diagramas de secuencia enfatizan cuándo ocurren estas interacciones.

En la figura 16.17 y 16.18 modelamos la secuencia de interacciones para una `Persona` que cambia de piso. La línea punteada que se extiende hacia abajo, desde el rectángulo de un objeto, es la *línea de vida* de ese objeto, la cual representa el paso del tiempo. Las acciones ocurren a lo largo de la línea de vida de un objeto, en orden cronológico de arriba hacia abajo; una acción cerca de la parte superior de una línea de vida ocurre antes que una acción cerca de la parte inferior.

El paso de mensajes en los diagramas de secuencia es similar al de los diagramas de colaboración (sección 11.9). Una flecha que se extiende desde el objeto que envía el mensaje, hasta el objeto que lo recibe, representa un mensaje entre dos objetos. La punta de flecha apunta hacia una *activación* en la línea de vida del objeto receptor. Una activación, que se muestra como un rectángulo delgado, indica que un objeto controla el flujo del programa. Cuando un objeto devuelve el control, se extiende un mensaje de retorno (representado como una línea punteada con una punta de flecha) se extiende desde la activación del objeto que devuelve el control, hasta la activación del objeto que envió inicialmente el mensaje. Para evitar un desorden, omitimos las flechas de retorno de mensajes.

La secuencia en la figura 16.17 empieza cuando una `Persona` solicita el `Elevador` enviando un mensaje `oprimirBoton` al `Boton` en el `Piso`. A continuación, el `Boton` envía el mensaje `solicitarElevador` al `ConductoElevador`, el cual solicita el servicio del `Elevador`. Según lo indicado por la condición entre corchetes, la `Persona` oprime el `Boton` solamente si la `puertaPiso` no está abierta ya.

Observe la división en el flujo para el `Elevador`, después de que éste es solicitado; el flujo de ejecución depende de cuál `Piso` fue el que generó la solicitud. Si el `Elevador` está en un `Piso` distinto del piso en el que se hizo la solicitud, el `Elevador` debe viajar hacia el `Piso` en el que está esperando la `Persona`. Para ahorrar espacio, la nota que indica que el `Elevador` se mueve al otro `Piso` representa a esta secuencia. Cuando el `Elevador` viaja hacia el otro `Piso`, la activación ramificada se fusiona con la activación original en la línea de vida del `Elevador`, y éste envía un mensaje `elevadorLlego` al objeto `puertaElevador`. Si el `Elevador` se encuentra en el mismo `Piso` en el que se hizo la solicitud, el objeto `puertaElevador` envía inmediatamente un mensaje `elevadorLlego` al objeto `puertaElevador`. El objeto `puertaElevador` recibe el mensaje de llegada y abre la `Puerta` en el `Piso` al que va a llegar el elevador.

La `Persona` debe esperar a que la `Puerta` del `Piso` se abra antes de entrar al `Elevador`. Las interacciones entre los subprocesos `Persona` en la simulación, el `Elevador` y los objetos `Puerta` son bastante complejas. Los objetos `Persona` y `Elevador` se ejecutan cada uno en subprocesos de ejecución separados, por lo que cuando interactúan debemos asegurarnos que estén bien sincronizados, para que la simulación funcione correctamente. Para asegurar que la `Puerta` del `Piso` no cierre mientras que una `Persona` esté tratando de caminar para acercarse o alejarse del `Elevador`, requerimos que cada `Persona` tenga que esperar para un

acceso exclusivo a la puerta del piso. Esta espera se lleva a cabo invocando al método `wait` de `puertaPiso` en un ciclo `while` que termina cuando se abre la `puertaPiso`. Un asterisco enfrente de la condición [`!puertaPiso.estaPuertaAbierta()`] modela este ciclo. Cuando la `puertaPiso` se abre, invoca a su propio método `notify`, se notifica al subproceso `Persona` que está en espera y el método `wait` termina. Después la `Persona` adquiere implícitamente un bloqueo en la `puertaPiso` y se asegura de que esté abierta. La `Persona` entra al `Elevador` invocando al método `establecerUbicacion` de `Persona`, con el `Elevador` como argumento `Ubicacion`. Al mantener el bloqueo en la `puertaPiso`, la `Persona` asegura que esta puerta no se cerrará y por lo tanto, evita lesionarse. En la implementación en Java veremos que la `Persona` también adquiere un bloqueo en el `Elevador`. Este bloqueo no estará disponible para la `Persona` que espera sino hasta que el pasajero (si existe uno) haya salido del `Elevador`.[6] La secuencia continúa en la figura 16.18.

Una vez que la `Persona` entra al elevador, debe indicar al `Elevador` que se mueva hacia el piso opuesto. Modelamos esta secuencia en la figura 16.18. La `Persona` empieza oprimiendo el `Boton` del `Elevador`. En respuesta al mensaje `botonOprimido`, el `Elevador` empieza el proceso de viajar hacia el piso opuesto, invocando a su propio método `establecerMovimiento` con un argumento de `true`. A continuación, el `Elevador` cierra la `PuertaElevador`. Después de cinco segundos de viaje, el `Elevador` deja de moverse invocando al método `establecerMovimiento` con un argumento `false`, llama a su propio método `cambiarPisos` para reflejar la nueva ubicación de `Piso` del `Elevador`, y envía un mensaje `elevadorLlego` a la `puertaPiso`. En respuesta al mensaje `elevadorLlego`, la `puertaElevador` se abre a sí misma y a la `puertaPiso` (no se muestra en la figura 16.18).

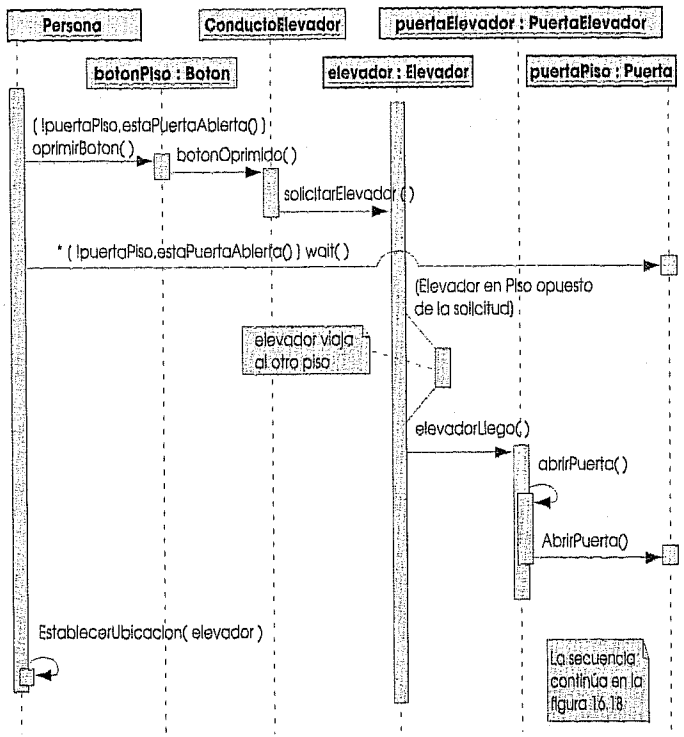


Figura 16.17 Diagrama de secuencia que modela a una `Persona` solicitando y entrando a un `Elevador`.

6. Hablaremos sobre los detalles de este bloqueo cuando veamos los detalles sobre la implementación en el apéndice E.

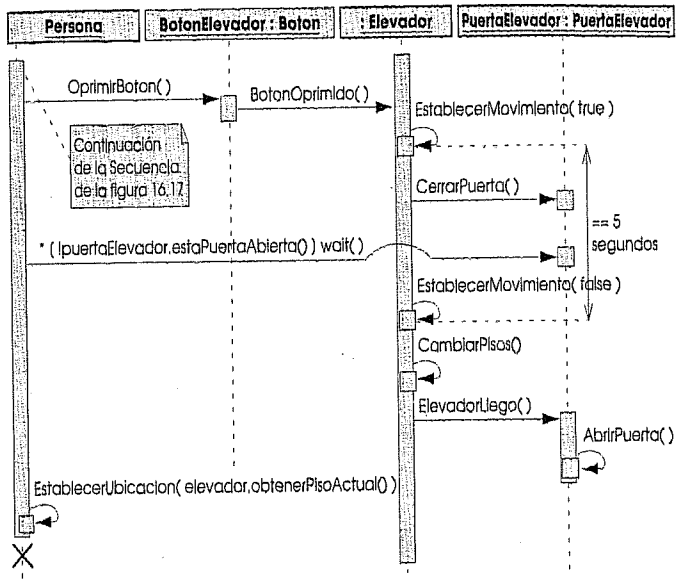


Figura 16.18 Diagrama de secuencia que modela a una `Persona` viajando en un `Elevador`.

La `Persona` que viaja en el `Elevador` debe esperar a que la `puertaPiso` se abra, por lo que la `Persona` envía un mensaje `wait` a la `puertaElevador`. Cuando la `puertaElevador` se abre en respuesta al mensaje `elevadorLlego`, la `puertaElevador` se notificará a sí misma (mediante `notify`), la invocación de `wait` de `Persona` terminará y la `Persona` comprobará si la `puertaElevador` está abierta. Después, la `Persona` establece su `Ubicacion` en el `Piso` de destino y sale de la simulación. Observe la "`X`" grande en la parte inferior de la línea de vida de la `Persona`. En un diagrama de secuencia, esta "`X`" indica que el objeto asociado se destruye a sí mismo. (En Java, el objeto `Persona` se convierte en candidato para la recolección de basura.)

Nuestro diagrama de clases final

La integración del subprocesamiento múltiple en nuestro modelo de elevador concluye el diseño de este modelo. Implementaremos este modelo en el apéndice E. En la figura 16.19 presentamos un diagrama de clases revisado. Observe que la principal diferencia entre los diagramas de clases de la figura 16.19 y la sección 11.31 es la presencia de las clases activas en la figura 16.19. Hemos establecido que las clases `Elevador` y `Persona` son clases activas. No obstante, el enunciado del problema mencionaba que "si una persona no entra ni solicita el elevador, éste cierra su puerta". Ya que hemos hablado sobre el subprocesamiento múltiple, creemos que un mejor requerimiento sería que los objetos `Puerta` se cerraran automáticamente (usando un subproceso) si han estado abiertas por más de un breve periodo (por ejemplo, tres segundos). Además, y aunque no se menciona en el enunciado del problema, los objetos `Luz` deben apagarse de manera automática; actualmente, los objetos `Luz` se apagan sólo cuando el `Elevador` sale de un `Piso`. Marcaremos las clases `Puerta` y `Luz` como clases activas para modelar este cambio. Implementaremos este cambio en el apéndice E.

En la figura 16.20 se presentan los atributos y las operaciones para todas las clases de la figura 16.19. Utilizaremos ambos diagramas para implementar el modelo. Además, hemos sustituido el método `partidaElevador` con el método `private establecerMovimiento`, ya que de acuerdo con la figura 16.18, el método `establecerMovimiento` proporciona el servicio que permite al `Elevador` partir de un `Piso`.

http://librosysolucionarios.net

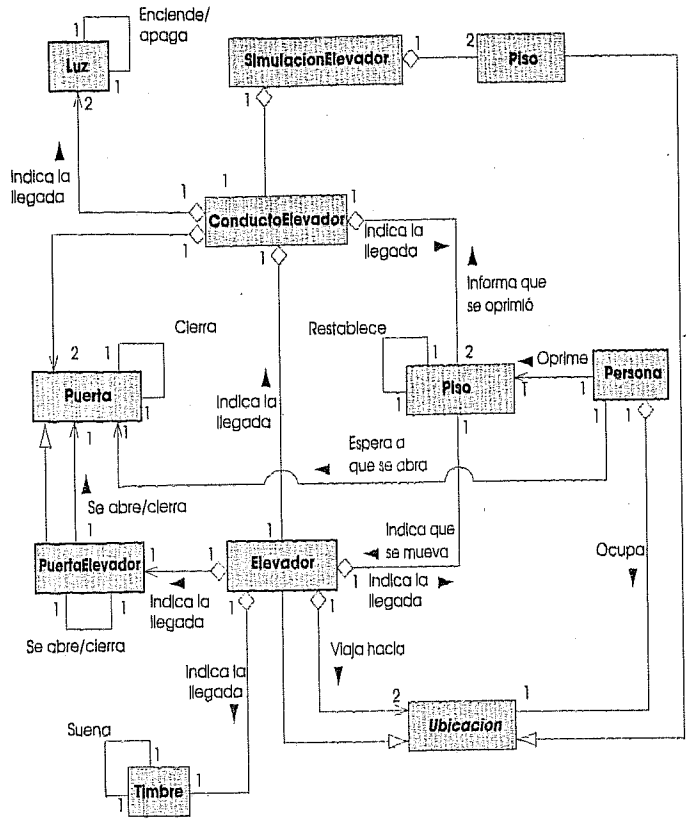


Figura 16.19 Diagrama de clases revisado de la simulación del elevador.

Utilizaremos estos diagramas de clases para implementar nuestro código de Java en el apéndice E, pero continuaremos haciendo pequeños ajustes "específicos de Java" en el código. En los apéndices, para cada clase, creamos métodos que acceden a las referencias a objetos y que implementan métodos de interfaz. En la sección 19.7 diseñaremos la vista: la presentación de nuestro modelo. Cuando implementemos esta presentación en el apéndice E, tendremos una simulación de elevador completamente funcional.

16.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño de concurrencia

Se han descubierto muchos patrones de diseño adicionales desde la publicación del libro de la Banda de los cuatro, el cual introdujo los patrones involucrados con los sistemas orientados a objetos. Algunos de estos nuevos patrones involucran a sistemas orientados a objetos específicos, como los sistemas concurrentes, paralelos o distribuidos. En esta sección hablaremos sobre los patrones de concurrencia para terminar con nuestra discusión acerca de la programación con subprocesamiento múltiple.

Patrones de diseño de concurrencia

Los lenguajes de programación con subprocesamiento múltiple como Java, permiten a los diseñadores especificar actividades concurrentes; es decir, aquéllas que operan en paralelo unas con otras. El diseño inapropiado

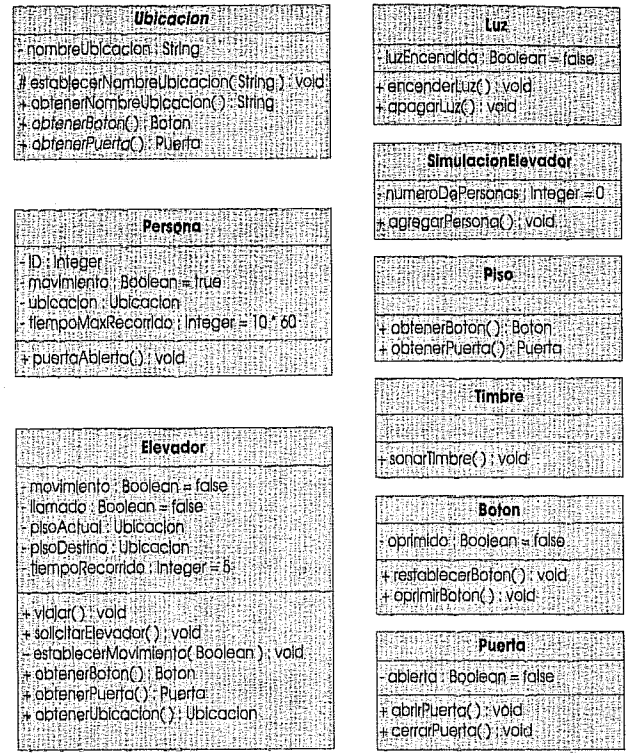


Figura 16.20 Diagrama de clases final con atributos y operaciones.

de los sistemas concurrentes puede ocasionar problemas de concurrencia. Por ejemplo, dos objetos que intenten alterar datos compartidos al mismo tiempo podrían corromper esos datos. Además, si dos objetos se esperan uno al otro a que terminen sus tareas, y si ninguno de los dos puede completar su tarea, estos objetos podrían, en un momento dado, esperar para siempre; una situación que se conoce como *interbloqueo*. Utilizando Java, Doug Lea[7] y Mark Grand[8] documentaron los *patrones de concurrencia* para las arquitecturas de diseño con subprocesamiento múltiple, para evitar varios problemas asociados con el subprocesamiento múltiple. A continuación proporcionamos una lista parcial de estos patrones de diseño:

- El *patrón de diseño Ejecución de un solo subproceso (Single-Threaded Execution)* (Grand, 2002) evita que varios subprocesos ejecuten concurrentemente el mismo método de otro objeto. En Java, la palabra clave `synchronized` (descrita en el capítulo 15) puede usarse para aplicar este patrón.
- El *patrón de diseño Suspensión precavida (Guarded Suspension)* (Lea, 2000) suspende la actividad de un subproceso y la reanuda cuando se cumpla cierta condición. En las líneas 137 a 147 y en las líneas 95 a 109 de `CaracteresAleatorios` (figura 16.16) se usa este patrón de diseño; los métodos `wait` y `notify` suspenden y reanudan los subprocesos del programa, y en la línea 98 se cambia alternadamente el valor de la variable de guardia que es evaluada por la condición.

7. Lea, D. *Concurrent Programing in Java, Second Edition: Design Principles and Patterns*. Massachusetts: Addison-Wesley, 2000.
8. Grand, M. *Patterns in Java; A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, Volume I*. Nueva York: John Wiley and Sons, 2002.

http://librosysolucionarios.net

- El *patrón de diseño Frustración (Balking)* (Lea, 2000) asegura que un método se *frustrará* (es decir, que regresará sin realizar ninguna acción) si un objeto ocupa un estado en el que no pueda ejecutar a ese método. Una variación de este patrón es que el método lanza una excepción describiendo por qué ese método no puede ejecutarlo; por ejemplo, un método que lanza una excepción al acceder a una estructura de datos que no existe.
- El *patrón de diseño Bloqueo de lectura/escritura (Read/Write Lock)* (Lea, 2000) permite a varios subprocesos obtener un acceso de lectura concurrente sobre un objeto, pero evita que varios subprocesos obtengan acceso de escritura concurrente en ese objeto. Sólo un subproceso a la vez puede obtener acceso de escritura a un objeto; cuando ese subproceso obtiene el acceso de escritura, el objeto está *bloqueado* para todos los demás subprocesos.
- El *patrón de diseño Término en dos fases (Two-Phase Termination)* (Grand, 98) utiliza un proceso de término en dos fases para un subproceso, para asegurar que el subproceso tenga la oportunidad de liberar recursos (como otros subprocesos engendrados) en memoria (fase uno) antes de que el subproceso termine (fase dos). En Java, un objeto `Thread` puede utilizar este patrón en el método `run`. Por ejemplo, el método `run` puede contener un ciclo infinito que se termina en base al cambio de cierto estado; al momento de terminar, el método `run` puede invocar un método `private` responsable de detener otros subprocesos engendrados (fase uno). A continuación, el subproceso termina después de que el método `run` termina (fase dos).

En la sección 18.12, "Descubrimiento de los patrones de diseño" regresaremos a los patrones de diseño de la Banda de los cuatro. Utilizando el material presentado en los capítulos 17 y 18, identificaremos esas clases en los paquetes `java.io` y `java.net` que utilizan patrones de diseño.

RESUMEN

- Las computadoras realizan operaciones en forma concurrente, como compilar programas, imprimir archivos y recibir mensajes de correo electrónico a través de una red.
- Los lenguajes de programación por lo general proporcionan sólo un simple conjunto de estructuras de control, las cuales permiten a los programadores realizar una acción a la vez y proceder a la siguiente acción.
- A través de la historia, la concurrencia se ha implementado con "primitivas" del sistema operativo disponibles sólo para los "programadores de sistemas".
- Java pone las primitivas de concurrencia a disposición del programador. El programador especifica que las aplicaciones contienen subprocesos de ejecución; cada subproceso designa una porción de un programa que pueda ejecutarse concurrentemente con otros subprocesos. A esto se le conoce como subprocesamiento múltiple.
- El método `run` de un método contiene el código que controla la ejecución de un subproceso.
- Un programa inicia la ejecución de un subproceso llamando al método `start` de ese subproceso, el cual establece la llamada inicial al método `run` del subproceso.
- El método `currentThread` de `Thread` devuelve una referencia al objeto `Thread` que se ejecuta en esos momentos.
- Un subproceso que acaba de crearse se encuentra en el estado *Nacimiento*. El subproceso permanece en este estado hasta que se hace una llamada al método `start` del subproceso; esto ocasiona que el subproceso entre al estado *Listo*.
- Un subproceso en el estado *Listo* entra al estado *En ejecución* cuando el sistema le asigna un procesador. El sistema asigna el procesador a uno de los subprocesos listos de mayor prioridad.
- Un subproceso entra al estado *Muerto* cuando termina su método `run`. El sistema, de un momento a otro, desechará un subproceso muerto.
- Un subproceso *En ejecución* entra al estado *Bloqueado* por una de varias razones, como cuando el subproceso envía una solicitud de entrada/salida. En este caso, el subproceso bloqueado pasa al estado *Listo* cuando se completa la acción de E/S que estaba esperando. Un subproceso bloqueado no puede usar un procesador, incluso si hay uno disponible.
- Si un subproceso llamó al método `wait`, la correspondiente llamada al método `notify` (si sólo hay un subproceso en espera) o `notifyAll` por aparte de otro subproceso en el programa pasará al subproceso original del estado *En espera* al estado *Listo*.
- El método `sleep` de `Thread` hace que un subproceso esté inactivo durante el número especificado de milisegundos. Un subproceso se despierta cuando expira el intervalo de inactividad designado.

- Si el subproceso 1 no puede continuar ejecutándose sino hasta que termine el subproceso 2, el subproceso 1 llama al método `join` del subproceso 2 para "unir" los dos subprocesos. Cuando se "unen" los dos subprocesos, el subproceso 1 sale del estado *En espera* cuando el subproceso 2 termina su ejecución (cuando entra al estado *Muerto*).
- En la sincronización de subprocesos, cuando un subproceso encuentra código que no puede ejecutar todavía, puede llamar al método `wait` para quedar suspendido, hasta que ocurran ciertas acciones que permitan al subproceso seguir ejecutándose.
- La espera puede ser peligrosa; puede provocar dos problemas severos conocidos como interbloqueo y aplazamiento indefinido. El aplazamiento indefinido también se conoce como inanición.
- Cualquier subproceso en el estado *En espera* o *Inactivo* puede salir de ese estado si otro subproceso invoca al método `interrupt` de `Thread` sobre ese subproceso en dicho estado.
- Todo subproceso en Java tiene una prioridad entre `MIN_PRIORITY` (1) y `MAX_PRIORITY` (10). De manera predeterminada, a cada subproceso se le asigna la prioridad `NORM_PRIORITY` (5). La prioridad de un subproceso puede ajustarse mediante el método `setPriority`. El método `getPriority` devuelve la prioridad de un subproceso.
- Algunas plataformas de Java soportan la partición de tiempo. Sin la partición de tiempo, un subproceso de igual prioridad se ejecuta hasta terminar, antes de que otros subprocesos de igual prioridad tengan la oportunidad de ejecutarse. Con la partición de tiempo, cada subproceso recibe un breve lapso del procesador, conocido como *cuanto*, durante el cual el subproceso puede ejecutarse. Al completarse el cuanto, incluso aunque el subproceso no haya terminado de ejecutarse, tal vez el procesador se quite de ese subproceso y se asigne al siguiente subproceso de igual prioridad, si hay uno disponible.
- El programador de subprocesos trata de mantener el subproceso de mayor prioridad *En ejecución* en todo momento y, si hay más de un proceso de mayor prioridad, debe asegurarse que todos los subprocesos con la misma alta prioridad se ejecuten durante un cuanto en forma cíclica (round-robin).
- Un subproceso puede llamar al método `yield` para dar a otros subprocesos de igual prioridad una oportunidad para ejecutarse.
- Cada objeto tiene un monitor. El monitor permite que sólo un subproceso a la vez ejecute una instrucción o método sincronizados en ese objeto.
- Un subproceso que ejecuta una instrucción sincronizada puede llegar a determinar que no puede continuar, por lo que el subproceso llama voluntariamente a `wait`. Esto evita que el subproceso compita por el procesador y lo coloca en el estado *En espera* de ese objeto.
- Un subproceso que ha llamado a `wait` es despertado por un subproceso que llama a `notify` o `notifyAll`. La notificación actúa como señal para el subproceso en espera, indicándole que la condición que ha estado esperando se ha cumplido (o podría cumplirse), por lo que es aceptable para ese subproceso tratar de volver a entrar al monitor.
- Un subproceso daemon sirve a otros subprocesos. Cuando sólo quedan subprocesos daemon en un programa, éste termina. Si un subproceso va a ser daemon, esto debe especificarse antes de iniciarlo.
- Para soportar el subprocesamiento múltiple en una clase que no sea subclase de `Thread`, debe implementar a la interfaz `Runnable` en esa clase.
- Un subproceso `Runnable` se crea pasando al constructor de la clase `Thread` una referencia a un objeto `Runnable`. El constructor de `Thread` especifica que el método `run` del objeto `Runnable` es el método que deberá invocarse cuando el subproceso empiece a ejecutarse.

TERMINOLOGÍA

a prueba de subprocesos
acceso a los datos compartidos con sincronización
adquirir el bloqueo de un objeto
aplazamiento indefinido
aplicación de un solo subproceso
Bloqueado, estado
bloqueo
bloqueo de entrada/salida
búfer circular
búfer compartido
ciclo de vida de un subproceso
cola de prioridades multinivel
Compleción de E/S
concurrencia
consumidor
cuanto
`currentThread`, método de `Thread`
Ejecutable, estado
En ejecución, estado
En espera, estado
estados de subproceso
fuga de memoria
`getName`, método de `Thread`
heredar prioridad de subproceso
`IllegalArgumentException`
`IllegalMonitorStateException`
`IllegalThreadStateException`
Inactivo, estado
inanición
instrucción sincronizada
interbloqueo
`interrupt`, método de `Thread`

http://librosysolucionarios.net

`InterruptedException`
`join`, método de `Thread`
Listo, estado
método sincronizado
Método `sleep( )` de `Thread`
monitor
Muerto, estado
`notify`, método de `Object`
`notifyAll`, método de `Object`
nuevo estado
partición de tiempo
prioridad de un subproceso
programa de un solo subproceso
programación cíclica (round-robin)
programación concurrente
programación preferente
programador
relación productor/consumidor
`run`, método de `Runnable`
`run`, método de un subproceso
`Runnable`, interfaz
`setDaemon`, método de `Thread`
`setPriority`, método de `Thread`
sincronización
sincronización de subprocesos
`sleep`, método de `Thread`
Solicitud de entrada/salida E/S
`start`, método de `Thread`
subprocesamiento múltiple
subproceso consumidor
subproceso daemon
subproceso de ejecución
subproceso productor
subproceso recolector de basura
subprocesos concurrentes asíncronos
subprocesos concurrentes productor y consumidor
`Thread`, clase
`Thread.MAX_PRIORITY`
`Thread.MIN_PRIORITY`
`Thread.NORM_PRIORITY`
`ThreadDeath`, excepción
variable de condición
`wait`, método de `Object`
`yield`, método de `Thread`

EJERCICIOS DE AUTOEVALUACIÓN

16.1 Complete las siguientes oraciones:

a) C y C++ son lenguajes con subprocesamiento _____, mientras que Java es un lenguaje con subprocesamiento _____.

b) Tres razones por las que un subproceso vivo podría no ser *Ejecutable* son que esté _____, _____ o _____.

c) Un subproceso entra al estado *Muerto* cuando _____.

d) La prioridad de un subproceso puede ajustarse mediante el método _____.

e) Un subproceso puede ceder el procesador a un subproceso de la misma prioridad mediante una llamada al método _____ de `Thread`.

f) Para detenerse durante cierto número designado de milisegundos y reanudar su ejecución, un subproceso debe llamar al método _____.

g) El método _____ pasa a un solo subproceso en el estado *En espera* de un objeto, al estado *Listo*.

h) El método _____ pasa a todos los subprocesos en el estado *En espera* de un objeto, al estado *Listo*.

16.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Un subproceso no es *Ejecutable* si está muerto.

b) En Java, un subproceso *Ejecutable* de mayor prioridad debe desplazar a los subprocesos de menor prioridad.

c) Algunos sistemas operativos utilizan la partición de tiempo con subprocesos. Por lo tanto, pueden permitir que los subprocesos desplacen a otros subprocesos de la misma prioridad.

d) Los subprocesos pueden ceder (`yield`) ante los subprocesos de menor prioridad.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

16.1 a) Simple, múltiple. b) *En espera*, *Inactivo*, *Bloqueado* para entrada/salida. c) Termina su método `run`. d) `setPriority`. e) `yield`. f) `sleep`. g) `notify`. h) `notifyAll`.

16.2 a) Verdadero. b) Verdadero. c) Falso. La partición de tiempo permite a un subproceso ejecutarse hasta que expira su porción de tiempo (o cuanto). Después pueden ejecutarse otros subprocesos de igual prioridad. d) Falso. Los subprocesos sólo pueden ceder ante los subprocesos de igual prioridad.

EJERCICIOS

16.3 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) El método `sleep` no consume tiempo del procesador cuando un subproceso está inactivo.

b) Al declarar un método como `synchronized` se garantiza que no pueda ocurrir el interbloqueo.

c) Los métodos `suspend`, `resume` y `stop` de `Thread` están desaprobados.

16.4 Defina cada uno de los siguientes términos.

a) subproceso,

b) subprocesamiento múltiple,

c) estado *Listo*,

d) Estado *Bloqueado*,

e) programación preferente,

f) Interfaz `Runnable`,

g) monitor,

h) método `notify`,

i) relación productor/consumidor.

16.5 Enliste cada una de las razones que se dan en el texto para entrar al estado *Bloqueado*. Para cada una de ellas, describa cómo saldría el programa normalmente del estado *Bloqueado* para entrar al estado *Listo*.

16.6 ¿Qué es la partición de tiempo? Indique una diferencia fundamental acerca de cómo se realiza la programación en sistemas Java que soportan la partición de tiempo, en comparación con la programación en sistemas Java que no soportan la partición de tiempo. ¿Por qué un subproceso llamaría a `yield`?

16.7 Indique la diferencia entra cada uno de los siguientes medios para detener subprocesos:

a) inactividad,

b) bloqueo de E/S.

16.8 Describa cada uno de los siguientes términos en el contexto de los mecanismos de subprocesamiento en Java:

a) monitor,

b) productor,

c) consumidor,

d) `wait`,

e) `notify`,

f) `InterruptedException`,

g) `synchronized`.

16.9 Escriba un programa en Java para demostrar que, a medida que se ejecute un subproceso de alta prioridad, éste retrasará la ejecución de todos los subprocesos de menor prioridad.

16.10 Si su sistema soporta la partición de tiempo, escriba un programa en Java para demostrar el uso de esta característica, entre varios subprocesos de igual prioridad. Muestre que la ejecución de un subproceso de menor prioridad se retrasa debido a la partición de tiempo de los subprocesos de mayor prioridad.

16.11 Escriba un programa en Java para demostrar cómo un subproceso de alta prioridad usa a `sleep` para dar a los subprocesos de menor prioridad la oportunidad de ejecutarse.

16.12 Si su sistema no soporta la partición de tiempo, escriba un programa en Java para demostrar cómo dos subprocesos usan `yield` para permitir cada uno que se ejecute el otro.

16.13 Dos problemas que pueden ocurrir en sistemas como Java (que permiten a los subprocesos esperar) son el interbloqueo, en el cual uno o más subprocesos esperarán para siempre un evento que no puede ocurrir, y el aplazamiento indefinido, en donde uno o más subprocesos se retrasarán durante cierto tiempo, sin saber cuánto. Dé un ejemplo de cómo estos problemas pueden ocurrir en un programa de Java con subprocesamiento múltiple.

16.14 (*Lectores y escritores.*) En este ejercicio usted deberá desarrollar un monitor de Java para resolver un famoso problema en el control de la concurrencia. Este problema fue discutido y resuelto por primera vez, por P. J. Courtois, F. Heymans y D. L. Parnas en su artículo de investigación, " Concurrent Control with Readers and Writers", *Communications of the ACM*, Vol. 14, No. 10, octubre 1971, pp. 667-668. El estudiante interesado podría también leer el artículo de investigación seminal de C. A. R. Hoare acerca de los monitores, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17, No. 10, octubre 1974, pp. 549 a 557. Corrigendum, *Communications of the ACM*, Vol. 18, No. 2, febrero 1975, p. 95. [El problema de los Lectores y escritores se describe detalladamente en el capítulo 5 del libro del autor: Deitel, H. M., *Operating Systems*, Reading, MA: Addison-Wesley, 1990.]

a) Con el subprocesamiento múltiple, muchos subprocesos pueden utilizar objetos compartidos; como hemos visto, el acceso a los objetos compartidos necesita sincronizarse cuidadosamente para evitar corromper los objetos.

b) Considere un sistema de reservación de una aerolínea, en el que muchos clientes tratan de reservar asientos en ciertos vuelos entre ciertas ciudades. Toda la información acerca de vuelos y asientos se almacena en una base de datos común en la memoria. Esta base de datos consiste de muchas entradas, en donde cada una de ellas representa a un asiento en cierto vuelo, para un día específico y entre ciertas ciudades específicas. En un escenario típico de reservación de aerolínea, el cliente navega por la base de datos, buscando el vuelo "óptimo" que cumpla con sus necesidades. Por lo tanto, un cliente puede llegar a sondear la base de datos muchas veces antes de decidir reservar un vuelo específico. Un asiento que haya estado disponible durante esta fase de sondeo podría fácilmente haber sido reservado por otra persona, antes de que el cliente tuviera la oportunidad de reser-

http://librosysolucionarios.net

varlo. En ese caso, cuando el cliente trate de hacer la reservación, descubrirá que los datos han cambiado y que el vuelo ya no está disponible.

c) Al cliente que sondea la base de datos se le conoce como un lector. Al cliente que trata de reservar el vuelo se le conoce como un escritor. Evidentemente, cualquier número de lectores pueden estar sondeando el objeto compartido a la vez, pero cada escritor necesita acceso exclusivo al objeto compartido, para evitar que éste se vuelva corrupto.

d) Escriba un programa en Java con subprocesamiento múltiple, que inicie varios subprocesos lector y varios subprocesos escritor, en donde cada uno de ellos trate de acceder a un solo registro de reservación. Un subproceso escritor tiene dos posibles transacciones: `hacerReservacion` y `cancelarReservacion`. Un lector tiene una posible transacción: `consultarReservacion`.

e) Primero implemente una versión de su programa que permita el acceso desincronizado al registro de reservación. Muestre cómo la integridad de la base de datos puede volverse corrupta. Después implemente una versión de su programa que utilice la sincronización del monitor de Java con `wait` y `notify`, para imponer un protocolo disciplinado para los lectores y escritores que acceden a los datos compartidos sobre las reservaciones. En especial, su programa debe permitir que varios lectores accedan al objeto compartido en forma simultánea cuando no haya un escritor activo. Pero si un escritor está activo, entonces no se debe permitir a ningún lector el acceso al objeto compartido.

f) Tenga cuidado. Este problema tiene muchas sutilezas. Por ejemplo, ¿qué ocurre cuando hay varios lectores activos y un escritor desea escribir? Si permitimos a un flujo estable de lectores que lleguen y compartan el objeto, podrían aplazar indefinidamente al escritor (quien se cansaría de esperar e iría a otra aerolínea a reservar un lugar). Para resolver este problema, tal vez sea conveniente optar por favorecer a los escritores. Pero aquí también hay una trampa, ya que un flujo estable de escritores podría entonces aplazar infinitamente a los lectores en espera y ellos, también, podrían optar por ir a otra aerolínea. Implemente su monitor con los siguientes métodos: `empezarALeer`, que será llamado por cualquier lector que desee tener acceso a una reservación; `dejarDeLeer`, que será llamado por cualquier lector que haya terminado de leer una reservación; `empezarAEscribir`, que será llamado por cualquier escritor que desee hacer una reservación; y `dejarDeEscribir`, que será llamado por cualquier escritor que haya terminado de hacer una reservación.

16.15 Escriba un programa para rebotar una pelota azul dentro de un subprograma. La pelota deberá empezar a moverse con un evento `mousePressed`. Cuando la pelota pegue en el borde del subprograma, deberá rebotar y continuar en la dirección opuesta.

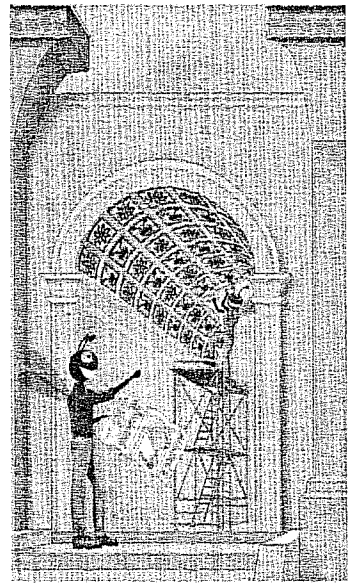
16.16 Modifique el programa del ejercicio 16.15 para agregar una nueva pelota cada vez que el usuario haga clic en el ratón. Proporcione un mínimo de 20 pelotas. Seleccione al azar el color para cada nueva pelota.

16.17 Modifique el programa del ejercicio 16.16 para agregar sombras. A medida que se mueva una pelota, dibuje un óvalo relleno de color negro en la parte inferior del subprograma. Tal vez sería conveniente agregar un efecto tridimensional, incrementando o decrementando el tamaño de cada pelota cuando ésta pegue en el borde del subprograma.

16.18 Modifique el programa de los ejercicios 16.15 o 16.16 para que las pelotas reboten cuando choquen unas con otras.

17

Archivos y flujos



Objetivos

- Crear, leer, escribir y actualizar archivos.
- Utilizar la clase `File`.
- Comprender la jerarquía de clases de flujos en Java.
- Utilizar las clases `FileInputStream` y `FileOutputStream`.
- Utilizar un cuadro de diálogo `JFileChooser` para tener acceso a los archivos y directorios.
- Utilizar las clases `ObjectInputStream` y `ObjectOutputStream`.
- Utilizar la clase `RandomAccessFile`.
- Familiarizarse con el procesamiento de archivos con acceso secuencial y acceso aleatorio.

Sólo puedo suponer que un documento "No archivar" se archiva en un archivo "No archivar".
Senador Frank Church
Audiencia del subcomité de inteligencia del Senado, 1975

La conciencia ... no aparece a sí misma cortada en pequeños pedazos. ... Un "río" o un "flujo" son las metáforas por las cuales se describe con más naturalidad.
William James

Leí una parte en su totalidad.
Samuel Goldwyn

Podría decirse que es un problema de tres pipas.
Sir Arthur Conan Doyle

http://librosysolucionarios.net

Plan general

17.1 Introducción
17.2 Jerarquía de datos
17.3 Archivos y flujos
17.4 La clase `File`
17.5 Creación de un archivo de acceso secuencial
17.6 Cómo leer datos desde un archivo de acceso secuencial
17.7 Actualización de archivos de acceso secuencial
17.8 Archivos de acceso aleatorio
17.9 Creación de un archivo de acceso aleatorio
17.10 Cómo escribir datos en forma aleatoria, en un archivo de acceso aleatorio
17.11 Cómo leer datos en forma secuencial, de un archivo de acceso aleatorio
17.12 Ejemplo práctico: Un programa para procesar transacciones
17.13 Nuevas APIs de E/S para la plataforma Java
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

17.1 Introducción

El almacenamiento de datos en variables y arreglos es temporal; los datos se pierden cuando una variable local queda fuera de alcance, o cuando el programa termina. Las computadoras utilizan *archivos* para la retención a largo plazo de grandes cantidades de datos, incluso hasta después de que terminan los programas que crean esos datos. Nos referimos a los datos que se mantienen en archivos como *datos persistentes*, ya que existen más allá de la duración de la ejecución del programa. Las computadoras almacenan archivos en *dispositivos de almacenamiento secundario* como discos magnéticos, discos ópticos y cintas magnéticas. En este capítulo explicaremos cómo los programas en Java crean, actualizan y procesan archivos de datos. Consideraremos tanto a los archivos de acceso secuencial como a los archivos de acceso aleatorio, y veremos las aplicaciones típicas para cada uno de ellos. También introduciremos las Nuevas APIs de E/S que se incluyen en la J2SE 1.4. En el capítulo 23, Conectividad de bases de datos en Java con JDBC™, continuaremos nuestra discusión acerca de la retención de datos a largo plazo mediante la creación de programas que interactúen con bases de datos.

El procesamiento de archivos es una de las herramientas más importantes que debe tener un lenguaje para soportar las aplicaciones comerciales, que generalmente procesan cantidades masivas de datos persistentes. En este capítulo hablaremos sobre las poderosas características de procesamiento de archivos y flujos de entrada/salida de Java. El procesamiento de archivos es un subconjunto de las herramientas para procesar flujos de Java, las cuales permiten a un programa leer y escribir bytes en memoria, en archivos y a través de conexiones de red. Tenemos dos metas en este capítulo: introducir los paradigmas del procesamiento de archivos y proporcionar al lector las suficientes herramientas de procesamiento de archivos como para soportar las características de red que se presentan en el capítulo 18.



Observación de ingeniería de software 17.1

Sería peligroso permitir que los subprogramas (applets) llegaran de cualquier parte del mundo en la World Wide Web y que pudieran leer y escribir en archivos del sistema cliente. De manera predeterminada, los navegadores Web evitan que los subprogramas lleven a cabo el procesamiento de archivos en el sistema cliente. Por lo tanto, los programas de procesamiento de archivos se implementan generalmente como aplicaciones de Java.[1]

1. Para obtener información sobre cómo utilizar archivos desde un subprograma, visite la página `http://java.sun.com/developer/technicalArticles/Security/Signed`.

http://libros-y-soluciones.net

17.2 Jerarquía de datos

Básicamente, una computadora procesa todos los elementos de datos como combinaciones de ceros y unos, ya que para los ingenieros es sencillo y económico el construir dispositivos electrónicos que puedan suponer dos estados estables: uno representa 0 y el otro, 1. Es increíble que las impresionantes funciones realizadas por las computadoras impliquen solamente las manipulaciones más fundamentales de 0s y 1s.

El elemento más pequeño de datos en una computadora puede asumir el valor 0 o 1. Dicho elemento de datos se conoce como *bit* (abreviatura de "dígito binario"); un dígito que puede suponer uno de dos valores). Los circuitos de computadora realizan varias manipulaciones simples de bits, como examinar o establecer el valor de un bit, o invertir su valor (de 1 a 0 o de 0 a 1).

Es muy difícil para los programadores trabajar con datos en el formato de bits de bajo nivel. En vez de ello, los programadores prefieren trabajar con datos en formatos como *dígitos decimales* (0-9), *letras* (A-Z y a-z) y *símbolos especiales* (por ejemplo, $, @, %, &, *, (,), -, +, ", :, ? y /). Los dígitos, letras y símbolos especiales se conocen como *caracteres*. El *conjunto de caracteres* de la computadora es el conjunto de todos los caracteres utilizados para escribir programas y representar elementos de datos. Las computadoras pueden procesar solamente 1s y 0s, por lo que un conjunto de caracteres representa a todos los caracteres como un patrón de 1s y 0s. Los caracteres en Java son caracteres *Unicode*, compuestos de dos *bytes*. Cada byte está compuesto de ocho bits. Vea el apéndice G para obtener más información acerca de Unicode.

Así como los caracteres están compuestos de bits, los *campos* están compuestos de caracteres o bytes. Un campo es un grupo de caracteres o bytes que transmiten cierto significado. Por ejemplo, un campo que consiste de letras mayúsculas y minúsculas puede utilizarse para representar el nombre de una persona.

Los elementos de datos que son procesados por las computadoras forman una *jerarquía de datos*, en la cual los elementos de datos se hacen más grandes y complejos en estructura, a medida que progresamos de bits a caracteres, de caracteres a campos, etcétera.

Generalmente, varios campos forman un *registro* (que se implementa como `class` en Java). Por ejemplo, en un sistema de nóminas un registro para un empleado específico podría estar compuesto de los siguientes campos (los posibles tipos para estos campos se muestran entre paréntesis, después de cada campo):

- Número de identificación del empleado (`int`).
- Nombre (`String`).
- Dirección (`String`).
- Sueldo por hora (`double`).
- Número de exenciones reclamadas (`int`).
- Ingresos desde inicio de año a la fecha (`int` o `double`).
- Monto de impuestos retenidos (`int` o `double`).

Por lo tanto, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada uno de los campos pertenece al mismo empleado. Desde luego que una compañía específica podría tener muchos empleados y, por ende, tendría un registro de nómina para cada empleado. Un *archivo* es un grupo de registros relacionados.[2] El archivo de nómina de una compañía generalmente contiene un registro para cada empleado. Por ejemplo, un archivo de nómina para una pequeña compañía podría contener sólo 22 registros, mientras que un archivo de nómina para una compañía grande podría contener 100,000 registros. Es común para una compañía tener muchos archivos, algunos de ellos conteniendo miles de millones, o incluso billones de caracteres de información. En la figura 17.1 se muestra la *jerarquía de datos*.

Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse cuando menos un campo en cada registro como *clave de registro*. Una clave de registro sirve para identificar que un registro pertenece a una persona o entidad específica, y es única en cada registro. Este campo generalmente se utiliza para buscar y ordenar registros. En el registro de nómina que describimos anteriormente, por lo general se elegiría el número de identificación de empleado como clave de registro.

2. Dicho en forma más general, un archivo contiene datos arbitrarios en formatos arbitrarios. A nivel del sistema operativo, un archivo se ve simplemente como una colección de bytes. Cualquier organización de los bytes en un archivo (como organizar los datos en registros) es una vista creada por el programador de aplicaciones.

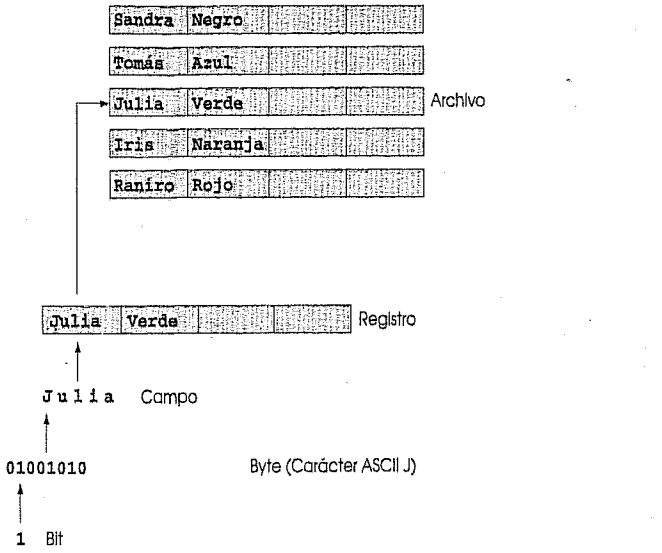


Figura 17.1 Jerarquía de datos.

Existen muchas formas de organizar los registros en un archivo. La organización más común se conoce como *archivo secuencial*, en el cual los registros se almacenan en orden, en base al campo que es la clave de registro. En un archivo de nómina, los registros se colocan en orden, en base al número de identificación de empleado. El registro del primer empleado en el archivo contiene el número de identificación más pequeño del empleado, y los registros siguientes contienen número de identificación de empleado cada vez mayores.

La mayoría de las empresas almacena datos en muchos archivos distintos. Por ejemplo, las compañías podrían tener archivos de nómina, de cuentas por cobrar (listas del dinero que deben los clientes), de cuentas por pagar (listas del dinero que se debe a los proveedores), archivos de inventarios (listas de información acerca de los artículos que maneja la empresa) y muchos otros tipos de archivos. A menudo, a un grupo de archivos relacionados se le conoce como *base de datos*. A una colección de programas diseñada para crear y administrar bases de datos se le conoce como *sistema de administración de bases de datos* (DBMS). Hablaremos sobre las bases de datos en el capítulo 23.

17.3 Archivos y flujos

Java considera a cada archivo como un *flujo* secuencial de bytes (figura 17.2). Cada sistema operativo proporciona un mecanismo para determinar el fin de un archivo, como el *marcador de fin de archivo* o la cuenta de bytes totales en el archivo que se registra en una estructura de datos administrativa, mantenida por el sistema. Java abstrae este concepto del programador. Un programa de Java que procesa un flujo de bytes simplemente recibe una indicación del sistema operativo cuando el programa llega al fin del flujo; el programa no necesita saber cómo representa la plataforma subyacente a los archivos o flujos. En algunos casos, la indicación de fin



Figura 17.2 La manera en que Java ve a un archivo de *n* bytes.

http://librosysolucionarios.net

de archivo ocurre como una excepción. En otros casos, la indicación es un valor de retorno de un método invocado en un objeto procesador de flujos. Demostraremos ambos casos en este capítulo.

Un programa de Java *abre* un archivo creando un objeto y asociándole un flujo de bytes. Java también puede asociar flujos de bytes con distintos dispositivos. De hecho, Java crea tres objetos flujo que se asocian con dispositivos cuando un programa de Java empieza a ejecutarse: `System.in`, `System.out` y `System.err`. El objeto `System.in` (el *objeto flujo de entrada estándar*) generalmente permite a un programa recibir bytes desde el teclado; el objeto `System.out` (el *objeto flujo estándar de salida*) generalmente permite a un programa mostrar datos en la pantalla; y el objeto `System.err` (el *objeto flujo estándar de error*) generalmente permite a un programa mostrar mensajes en la pantalla. Cada uno de estos flujos puede *redirigirse*. Para `System.in`, esta capacidad permite al programa leer bytes desde un origen distinto. Para `System.out` y `System.err`, esta capacidad permite que la salida se envíe a una ubicación distinta, como un archivo en disco. La clase `System` proporciona los métodos `setIn`, `setOut` y `setErr` para redirigir los flujos estándar de entrada, salida y error, respectivamente.

Los programas de Java realizan el procesamiento de archivos utilizando clases del paquete `java.io`. Este paquete incluye definiciones para las clases de flujo como `FileInputStream` (para la entrada basada en bytes desde un archivo), `FileOutputStream` (para la salida basada en bytes hacia un archivo), `FileReader` (para la entrada basada en caracteres desde un archivo) y `FileWriter` (para la salida basada en caracteres hacia un archivo). Los archivos se abren creando objetos de estas clases de flujos, que heredan de las clases `InputStream`, `OutputStream`, `Reader` y `Writer`, respectivamente. Por lo tanto, los métodos de estas clases de flujos pueden aplicarse a los flujos de archivos también. Para realizar operaciones de entrada y salida con tipos de datos, se utilizarán objetos de las clases `ObjectInputStream`, `DataInputStream`, `ObjectOutputStream` y `DataOutputStream` junto con las clases de flujos de archivos basadas en bytes `FileInputStream` y `FileOutputStream`. La jerarquía de clases en el paquete `java.io` puede consultarse en la documentación en línea, en la página:

```
java.sun.com/j2se/1.4.1/docs/api/java/io/package-tree.html
```

En la jerarquía, cada nivel de sangría indica que la clase con sangría extiende a la clase encima de ella. Por ejemplo, la clase `InputStream` es una subclase de `Object`. Haga clic en el nombre de una clase en la jerarquía para ver los detalles de esa clase.

Como puede ver en la jerarquía, Java ofrece muchas clases para realizar operaciones de entrada/salida. En esta sección presentaremos una breve descripción general de estas clases, y explicaremos cómo se relacionan entre sí. Utilizaremos varias de estas clases de flujos en este capítulo, a medida que vayamos implementando programas de procesamiento de archivos para crear, manipular y destruir archivos de acceso secuencial y archivos de acceso aleatorio. También incluiremos un ejemplo detallado acerca de la clase `File`, que es útil para obtener información sobre archivos y directorios. En el capítulo 18, Redes, utilizaremos las clases de flujos en forma extensa, para implementar aplicaciones de red.

Clases de entrada y salida basadas en bytes

`InputStream` y `OutputStream` (subclases de `Object`) son clases `abstract` que declaran métodos para realizar operaciones basadas en bytes de entrada y salida, respectivamente. En este capítulo utilizaremos en forma extensa las clases `FileInputStream` (una subclase de `InputStream`) y `FileOutputStream` (una subclase de `OutputStream`) para manipular archivos.

Las canalizaciones son canales de comunicación sincronizados entre subprocesos o procesos. Java proporciona las clases `PipedOutputStream` (una subclase de `OutputStream`) y `PipedInputStream` (una subclase de `InputStream`) para establecer canalizaciones entre dos subprocesos. Un subproceso envía datos a otro, escribiendo a un objeto `PipedOutputStream`. El subproceso de destino lee la información de la canalización mediante un objeto `PipedInputStream`.

Un objeto `FilterInputStream` *filtra* a un objeto `InputStream`, y un objeto `FilterOutputStream` filtra a un objeto `OutputStream`; filtrar significa simplemente que el flujo que actúa como filtro proporciona una funcionalidad adicional, como un búfer, el monitoreo de los números de línea o la agregación de bytes de datos en unidades de tipo primitivo significativas. `FilterInputStream` y `FilterOutputStream` son clases `abstract`, por lo que sus subclases proporcionan una funcionalidad adicional.

Un objeto `PrintStream` (una subclase de `FilterOutputStream`) envía texto como salida hacia el flujo especificado. En realidad, hemos estado utilizando la salida mediante `PrintStream` a lo largo de este texto, hasta este punto; `System.out` es un objeto `PrintStream`, al igual que `System.err`.

Leer datos en forma de bytes sin ningún formato es un proceso rápido, pero crudo. Por lo general, los programas leen datos como agregados de bytes que forman un valor `int`, un `float`, un `double` y así, sucesivamente. Los programas de Java pueden utilizar varias clases para recibir datos de entrada y enviar datos de salida en forma de agregación.

Un objeto `RandomAccessFile` es útil para las aplicaciones de acceso directo, como las aplicaciones para procesar transacciones (por ejemplo, sistemas de reservaciones de aerolíneas y sistemas de punto de venta). Con un archivo de acceso secuencial, cada solicitud de entrada/salida sucesiva lee o escribe el siguiente conjunto consecutivo de datos en el archivo. Con un archivo de acceso aleatorio, cada solicitud de entrada/salida sucesiva podría dirigirse a cualquier parte del archivo; tal vez una sección que esté ampliamente separada de la parte del archivo a la que se hizo referencia en la solicitud anterior. Las aplicaciones de acceso directo proporcionan un rápido acceso a los elementos de datos específicos en archivos extensos; a menudo, dichas aplicaciones requieren que los usuarios esperen las respuestas; éstas deben estar disponibles rápidamente, o las personas podrían impacientarse y buscar en otro lugar.

La interfaz `DataInput` describe métodos para leer tipos primitivos desde un flujo de entrada. Las clases `DataInputStream` y `RandomAccessFile` implementan a esta interfaz para leer conjuntos de bytes y verlos como valores de tipo primitivo. La interfaz `DataInput` incluye los métodos `readLine` (para arreglos `byte`), `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (para arreglos `byte`), `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (para cadenas) y `skipBytes`.

La interfaz `DataOutput` describe un conjunto de métodos para escribir tipos primitivos hacia un flujo de salida. Las clases `DataOutputStream` (una subclase de `FilterOutputStream`) y `RandomAccessFile` implementan a esta interfaz para escribir valores de tipos primitivos como bytes. La interfaz `DataOutput` incluye los métodos `write` (para un `byte`), `write` (para un arreglo `byte`), `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars` (para objetos `String` Unicode), `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort` y `writeUTF`.

El *uso de un búfer* es una técnica para mejorar el rendimiento de las operaciones de E/S. Con un objeto `BufferedOutputStream` (una subclase de la clase `FilterOutputStream`), cada instrucción de salida no produce necesariamente una transferencia física real de datos hacia el dispositivo de salida. En vez de ello, cada operación de salida se dirige hacia una región en memoria conocida como búfer, que es lo suficientemente grande como para almacenar los datos de muchas operaciones de salida. Después, la transferencia real hacia el dispositivo de salida se realiza en una sola *operación física de salida* extensa cada vez que se llena el búfer. Las operaciones de salida dirigidas hacia el búfer de salida en memoria se conocen a menudo como *operaciones lógicas de salida*. Con un objeto `BufferedOutputStream`, se puede forzar a un búfer parcialmente lleno para que envíe su contenido al dispositivo en cualquier momento, mediante la invocación del método `flush` del objeto flujo.

Con un objeto `BufferedInputStream` (una subclase de la clase `FilterInputStream`), muchos trozos "lógicos" de datos de un archivo se leen como una sola *operación física de entrada* extensa y se envían a un búfer de memoria. A medida que un programa solicita cada nuevo trozo de datos, éste se toma del búfer. (A este procedimiento se le conoce como *operación lógica de entrada*). Cuando el búfer está vacío, se lleva a cabo la siguiente operación física de entrada real desde el dispositivo de entrada, para leer el siguiente grupo de trozos "lógicos" de datos. Por lo tanto, el número de operaciones físicas de entrada reales es pequeño, en comparación con el número de solicitudes de lectura emitidas por el programa.

Tip de rendimiento 17.1



Las operaciones de entrada/salida típicas son extremadamente lentas, en comparación con la velocidad de acceso de la memoria de la computadora. Las operaciones de entrada y salida con búfer normalmente producen mejoras considerables en el rendimiento, en comparación con las operaciones de entrada y salida sin búfer.

Cuando las variables de instancia se envían a un archivo en disco, perdemos la información de tipo del objeto, en cierto sentido. En un disco tenemos solamente datos, y no información sobre el tipo de cada dato. El programa que va a leer estos datos "sabe" a qué tipo de objeto corresponden estos datos, entonces simplemente se leen y

se colocan en objetos de ese tipo. Algunas veces es conveniente leer o escribir todo un objeto completo en un archivo. Las clases `ObjectInputStream` y `ObjectOutputStream`, que implementan respectivamente a las interfaces `ObjectInput` y `ObjectOutput`, permiten que se lea todo un objeto completo desde, o hacia, un archivo (u otro tipo de flujo). A menudo envolvemos los objetos `FileInputStream` en objetos `ObjectInputStream`. (También envolvemos los objetos `FileOutputStream` en objetos `ObjectOutputStream`.) La interfaz `ObjectOutput` contiene el método `writeObject`, el cual toma un objeto `Object` que implementa a la interfaz `Serializable` como un argumento y escribe su información en el objeto `OutputStream`. De manera correspondiente, la interfaz `ObjectInput` requiere el método `readObject`, el cual lee y devuelve un objeto `Object` de un objeto `InputStream`. Una vez que se ha leído un objeto, puede convertirse al tipo deseado. Además, estas interfaces incluyen otros métodos centrados en `Object`, así como los mismos métodos que `DataInput` y `DataOutput` para leer y escribir tipos primitivos.

La E/S de flujos en Java incluye herramientas para recibir datos de entrada de arreglos `byte` en memoria, y enviar datos de salida a arreglos `byte` en memoria. Un objeto `ByteArrayInputStream` (una subclase de `InputStream`) lee de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` (una subclase de `OutputStream`) escribe en un arreglo `byte` en memoria. Una aplicación de la E/S con arreglos `byte` es la validación de datos. Un programa puede recibir como entrada una línea completa a la vez desde el flujo de entrada, para colocarla en un arreglo `byte`. Después puede usarse una rutina de validación para analizar detalladamente el contenido del arreglo `byte` y corregir los datos, si es necesario. Finalmente, el programa puede recibir los datos de entrada del arreglo `byte`, "sabiendo" que los datos de entrada se encuentran en el formato adecuado. Enviar datos de salida a un arreglo `byte` es una excelente manera de aprovechar las poderosas herramientas de formato para los datos de salida que proporcionan los flujos en Java. Por ejemplo, los datos pueden prepararse en un arreglo `byte`, utilizando el mismo formato que se mostrará posteriormente, y se pueden enviar hacia un archivo en disco para preservar la imagen en pantalla.

Un objeto `SequenceInputStream` (una subclase de `InputStream`) permite la concatenación de varios objetos `InputStream`, por lo que el programa ve al grupo como un flujo `InputStream` continuo. Cuando el programa llega al final de un flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.

Flujos de entrada y salida basados en caracteres

Además de los flujos basados en caracteres, Java proporciona las clases `Reader` y `Writer`, que son flujos basados en caracteres Unicode de dos bytes. La mayoría de los flujos basados en caracteres tienen sus correspondientes clases `Reader` o `Writer` basadas en caracteres.

Las clases `BufferedReader` (una subclase de la clase `abstract Reader`) y `BufferedWriter` (una subclase de la clase `abstract Writer`) permiten el uso eficiente del búfer para los flujos basados en caracteres. Los flujos basados en caracteres utilizan caracteres Unicode; dichos flujos pueden procesar datos en cualquier lenguaje que sea representado por el conjunto de caracteres Unicode.

Las clases `CharArrayReader` y `CharArrayWriter` leen y escriben, respectivamente, un flujo de caracteres en un arreglo de caracteres. Un objeto `LineNumberReader` (una subclase de `BufferedReader`) es un flujo de caracteres con búfer que lleva el registro de los números de línea (es decir, una nueva línea, un retorno o una combinación de retorno de carro y avance de línea.

Las clases `FileReader` (una subclase de `InputStreamReader`) y `FileWriter` (una subclase de `OutputStreamWriter`) leen caracteres de, y escriben caracteres en, un archivo, respectivamente. Las clases `PipedReader` y `PipedWriter` implementan flujos de caracteres canalizados, que pueden utilizarse para transferir la información entre subprocesos. Las clases `StringReader` y `StringWriter` leen y escriben caracteres, respectivamente, en objetos `String`. Un objeto `PrintWriter` escribe caracteres en un flujo.

17.4 La clase `File`

En esta sección presentamos la clase `File`, que es especialmente útil para recuperar información acerca de un archivo o directorio de un disco. Los objetos de la clase `File` no abren archivos ni proporcionan herramientas para procesar archivos. No obstante, los objetos `File` se utilizan frecuentemente con objetos de otras clases de `java.io` para especificar los archivos o directorios que van a manipularse.

http://librosysolucionarios.net

La clase `File` proporciona cuatro constructores. El constructor:

```
public File( String nombre )
```

especifica el `nombre` de un archivo o directorio que se va a asociar con el objeto `File`. El `nombre` puede contener *información sobre la ruta*, así como el nombre del archivo o directorio. La ruta de un archivo o directorio especifica la ubicación del archivo o directorio en el disco. La ruta incluye algunos o todos los directorios que conducen a ese archivo o directorio. Una *ruta absoluta* contiene todos los directorios, empezando con el *directorio raíz*, que conducen a un archivo o directorio específico. Cada archivo o directorio en un disco duro específico tiene el mismo directorio raíz en su ruta. Una *ruta relativa* contiene un subconjunto de los directorios que conducen a un archivo o directorio específico. Las rutas relativas normalmente empiezan desde el directorio en el que la aplicación empezó a ejecutarse.

El constructor:

```
public File( String rutaAlNombre, String nombre )
```

usa el argumento `rutaAlNombre` (una ruta absoluta o relativa) para localizar el archivo o directorio especificado por `nombre`.

El constructor:

```
public File( File directorio, String nombre )
```

usa un objeto `File` existente llamado `directorio` (una ruta absoluta o relativa) para localizar el archivo o directorio especificado por `nombre`. En la figura 17.3 se describen algunos métodos comunes de `File`. La lista completa de métodos `File` puede verse en:

```
java.sun.com/j2se/1.4.1/docs/api/java/io/File.html.
```

El constructor:

```
public File( URI uri )
```

usa el objeto `URI` dado para localizar el archivo. Un *Identificador uniforme de recursos (URI)* es una forma más general de un *Localizador uniforme de recursos (URL)*, el cual se utiliza comúnmente para localizar sitios Web. Por ejemplo, `http://www.deitel.com/` es el URL para el sitio Web de Deitel & Associates. Los URIs para localizar archivos varían entre los distintos sistemas operativos. En plataformas Windows, el URI:

```
file:/C:/datos.txt
```

identifica al archivo `datos.txt`, almacenado en el directorio raíz de la unidad C:.



Buena práctica de programación 17.1

Use el método `isFile` de `File` para determinar que un objeto `File` represente a un archivo (y no a un directorio) antes de tratar de abrir un archivo.

Método	Descripción
`boolean canRead()`	Devuelve `true` si un archivo puede leerse; `false` en caso contrario.
`boolean canWrite()`	Devuelve `true` si puede escribirse en un archivo; `false` en caso contrario.
`boolean exists()`	Devuelve `true` si el nombre especificado como argumento para el constructor `File` es un archivo o directorio en la ruta especificada; `false` en caso contrario.
`boolean isFile()`	Devuelve `true` si el nombre especificado como argumento para el constructor `File` es un archivo; `false` en caso contrario.
`boolean isDirectory()`	Devuelve `true` si el nombre especificado como argumento para el constructor `File` es un directorio; `false` en caso contrario.

Figura 17.3 Los métodos de `File`. (Parte 1 de 2.)

Método	Descripción
`boolean isAbsolute()`	Devuelve `true` si los argumentos especificados para el constructor de `File` indican una ruta absoluta a un archivo o directorio; `false` en caso contrario.
`String getAbsolutePath()`	Devuelve una cadena con la ruta absoluta del archivo o directorio.
`String getName()`	Devuelve una cadena con el nombre del archivo o directorio.
`String getPath()`	Devuelve una cadena con la ruta del archivo o directorio.
`String getParent()`	Devuelve una cadena con el directorio padre del archivo o directorio; es decir, el directorio en el que puede encontrarse ese archivo o directorio.
`long length()`	Devuelve la longitud del archivo, en bytes. Si el objeto `File` representa a un directorio, se devuelve `0`.
`long lastModified()`	Devuelve una representación dependiente de la plataforma de la hora en la que se hizo la última modificación en el archivo o directorio. El valor devuelto es útil sólo para compararlo con otros valores devueltos por este método.
`String[] list()`	Devuelve un arreglo de cadenas, las cuales representan el contenido de un directorio. Devuelve `null` si el objeto `File` no es un directorio.

Figura 17.3 Los métodos de `File`. (Parte 2 de 2.)

En la figura 17.4 se demuestra el uso de la clase `File`. La aplicación `PruebaFile` crea una GUI que contiene un objeto `JTextField` para introducir un nombre de archivo o directorio, y un objeto `JTextArea` para mostrar información acerca del nombre de archivo o directorio que se introdujo.

```
1  // Fig. 17.4: PruebaFile.java
2  // Demostración de la clase File.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import javax.swing.*;
7
8  public class PruebaFile extends JFrame
9     implements ActionListener {
10
11    private JTextField campoEntrada;
12    private JTextArea areaSalida;
13
14    // configurar la GUI
15    public PruebaFile()
16    {
17       super( "Prueba de la clase File" );
18
19       campoEntrada = new JTextField( "Escriba aquí el nombre del archivo o directorio" );
20       campoEntrada.addActionListener( this );
21       areaSalida = new JTextArea();
22
23       ScrollPane panelDesplazable = new ScrollPane();
24       panelDesplazable.add( areaSalida );
25
26       Container contenedor = getContentPane();
27       contenedor.add( campoEntrada, BorderLayout.NORTH );
28       contenedor.add( panelDesplazable, BorderLayout.CENTER );
```

Figura 17.4 La clase `File` utilizada para obtener información de archivos y directorios. (Parte 1 de 3.)

http://librosysolucionarios.net

```
29
30          setSize( 400, 400 );
31          setVisible( true );
32
33       } // fin del constructor
34
35       // mostrar información acerca del archivo especificado por el usuario
36       public void actionPerformed( ActionEvent eventoAccion )
37       {
38          File nombre = new File( eventoAccion.getActionCommand() );
39
40          // si nombre existe, mostrar información sobre él
41          if ( nombre.exists() ) {
42             areaSalida.setText( nombre.getName() + " existe\n" +
43                ( nombre.isFile() ? "es un archivo\n" : "no es un archivo\n" ) +
44                ( nombre.isDirectory() ? "es un directorio\n" :
45                   "no es un directorio\n" ) +
46                ( nombre.isAbsolute() ? "es una ruta absoluta\n" :
47                   "no es una ruta absoluta\n" ) + "Última modificación: " +
48                nombre.lastModified() + "\nLongitud: " + nombre.length() +
49                "\nRuta: " + nombre.getPath() + "\nRuta absoluta: " +
50                nombre.getAbsolutePath() + "\nPadre: " + nombre.getParent() );
51
52             // mostrar información si nombre es un archivo
53             if ( nombre.isFile() ) {
54
55                // anexar el contenido del archivo a areaSalida
56                try {
57                   BufferedReader entrada = new BufferedReader(
58                      new FileReader( nombre ) );
59                   StringBuffer bufer = new StringBuffer();
60                   String texto;
61                   areaSalida.append( "\n\n" );
62
63                   while ( ( texto = entrada.readLine() ) != null )
64                      bufer.append( texto + "\n" );
65
66                   areaSalida.append( bufer.toString() );
67                }
68
69                // procesar los problemas en el procesamiento del archivo
70                catch( IOException excepcionES ) {
71                   JOptionPane.showMessageDialog( this, "ERROR EN ARCHIVO",
72                      "ERROR EN ARCHIVO", JOptionPane.ERROR_MESSAGE );
73                }
74
75             } // fin de instrucción if
76
77             // mostrar listado de directorio
78             else if ( nombre.isDirectory() ) {
79                String directorio[] = nombre.list();
80
81                areaSalida.append( "\n\nContenido del directorio:\n");
82
83                for ( int i = 0; i < directorio.length; i++ )
```

Figura 17.4 La clase `File` utilizada para obtener información de archivos y directorios. (Parte 2 de 3.)


```
84                areaSalida.append( directorio[ i ] + "\n" );
85             }
86
87          } // fin de instrucción if externa
88
89          // no es archivo ni directorio, mostrar mensaje de error
90          else {
91             JOptionPane.showMessageDialog( this,
92                eventoAccion.getActionCommand() + " no existe",
93                "ERROR", JOptionPane.ERROR_MESSAGE );
94          }
95
96       } // fin del método actionPerformed
97
98       public static void main( String args[] )
99       {
100         PruebaFile aplicacion = new PruebaFile();
101         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
102      }
103
104   } // fin de la clase PruebaFile
```

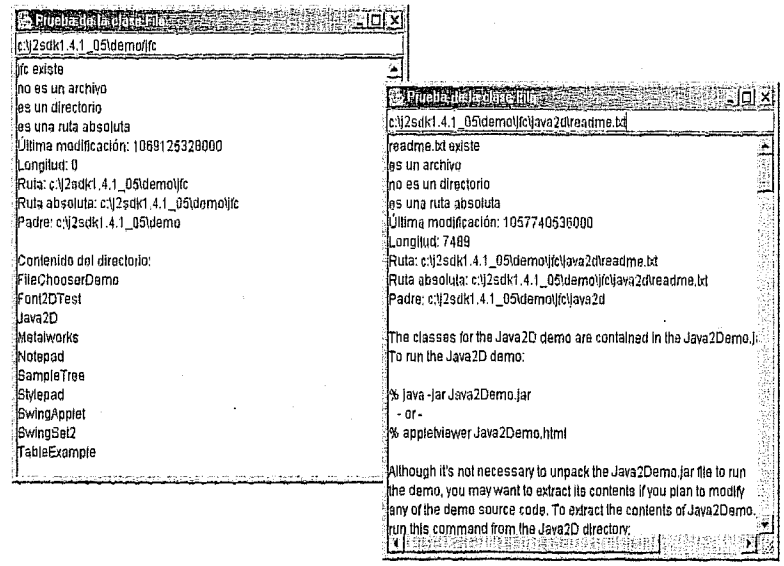


Figura 17.4 La clase `File` utilizada para obtener información de archivos y directorios. (Parte 3 de 3.)

El usuario escribe un nombre de archivo o directorio en el campo de texto y oprime la tecla *Intro* para invocar al método `actionPerformed` (líneas 36 a 96), el cual crea un nuevo objeto `File` (línea 38) y asigna su referencia a `nombre`. En la línea 41 se invoca el método `exists` de `File` para determinar si el nombre introducido por el usuario existe (ya sea como un archivo o directorio) en el disco. Si el nombre introducido por el usuario no existe, el método `actionPerformed` continúa con las líneas 90 a 94 y muestra un cuadro de diálogo de mensaje con el nombre que escribió el usuario, seguido de `"no existe"`. De no ser así, se ejecuta el cuerpo de la instrucción `if` (líneas 42 a 86). El programa muestra el nombre del archivo o directorio, seguido de los resultados obtenidos al probar el objeto `File` con los métodos `isFile` (línea 43), `isDirec-`

http://librosysolucionarios.net

`tory` (línea 44) e `isAbsolute` (línea 46). A continuación, el programa muestra los valores devueltos por `lastModified` (línea 48), `length` (línea 48), `getPath` (línea 49), `getAbsolutePath` (línea 50) y `getParent` (línea 50).

Si el objeto `File` representa a un archivo (línea 53), el programa lee el contenido de este archivo y lo muestra en el objeto `JTextArea` (líneas 56 a 73). Para los fines de este programa, nos gustaría leer líneas de texto de un archivo. La clase `FileReader` puede utilizarse para abrir un archivo para leer caracteres Unicode. Desafortunadamente, la clase `FileReader` no sabe cómo leer líneas de texto. La clase `BufferedReader` proporciona la funcionalidad de leer líneas de texto, pero no sabe cómo abrir un archivo en modo de entrada. Por lo tanto, necesitamos combinar la funcionalidad de estas dos clases. La solución a este problema es una técnica conocida como *envoltura de objetos flujo* (la habilidad de agregar los servicios de un flujo a otro). Para envolver un objeto `FileReader` en un objeto `BufferedReader`, pasamos el objeto `FileReader` al constructor de `BufferedReader` (líneas 57 y 58). El constructor podría lanzar una excepción `IOException` si ocurre un problema durante la apertura del archivo (por ejemplo, cuando se abre un archivo que no existe en modo de lectura). De ser así, el programa muestra un cuadro de diálogo con un mensaje de error (líneas 70 a 73). Si la construcción de los dos flujos no lanza una excepción `IOException`, el archivo se abre. A continuación, puede usarse la referencia `entrada` para leer del archivo. En la línea 59 se crea un objeto `StringBuffer` para almacenar el contenido del archivo en memoria. En las líneas 63 y 64 se utiliza el método `readLine` de `BufferedReader` para leer una línea a la vez, como una cadena, y para anexar esa cadena al objeto `StringBuffer`. El método `readLine` devuelve `null` cuando se llega al fin del archivo.

Si el objeto `File` representa a un directorio (línea 78), el programa lee su contenido mediante el uso del método `list` de `File` y muestra el contenido del directorio en el objeto `JTextArea`.

Los primeros resultados de este programa demuestran el uso de un objeto `File` asociado con el directorio `jfc` del Kit de desarrollo de software para Java 2. Los siguientes resultados demuestran el uso de un objeto `File` asociado con el archivo `readme.txt` del directorio de ejemplo Java 2D que viene con el Kit de desarrollo de software para Java 2. En ambos casos especificamos una ruta absoluta en nuestra computadora personal.

Un *carácter separador* se utiliza para separar directorios y archivos en la ruta. En un equipo Windows, el carácter separador es la barra diagonal inversa (\ ). En una estación de trabajo UNIX, el carácter separador es la barra diagonal (/). Java procesa ambos caracteres en forma idéntica en el nombre de una ruta. Por ejemplo, aunque la salida de ejemplo utiliza la ruta:

```
c:\j2sdk1.4.1_05\demo/jfc
```

que emplea uno de cada uno de los caracteres separadores antes mencionados, Java de todas formas procesa la ruta apropiadamente.



Error común de programación 17.1

Usar \ como separador de directorios en vez de \\ en una literal de cadena es un error lógico. Una sola \ indica que la \ y el siguiente carácter representan una secuencia de escape. Para insertar una \ en una literal de cadena, debe usar \\.



Buena práctica de programación 17.2

Al crear cadenas que representen información de rutas, use el método `File.pathSeparator` para obtener el carácter separador apropiado para el equipo local, en vez de utilizar explícitamente a / o \.

17.5 Creación de un archivo de acceso secuencial

Java no impone una estructura en un archivo, por lo que los conceptos como un registro no existen en los archivos de Java. Por lo tanto, los programadores deben estructurar los archivos de manera que cumplan con los requerimientos de sus aplicaciones. En el siguiente ejemplo veremos cómo el programador puede imponer una estructura de registros simple en un archivo. Primero presentaremos el programa; después lo analizaremos detalladamente.

El programa de las figuras 17.5 a 17.7 crea un archivo simple de acceso secuencial, que podría utilizarse en un sistema de cuentas por cobrar para ayudar a administrar el dinero que deben a una compañía los clientes a crédito. Por cada cliente, el programa obtiene un número de cuenta, el primer nombre del cliente, su apellido y su saldo (es decir, el monto que el cliente aún debe a la compañía por los bienes y servicios recibidos en el pasado). Los datos obtenidos para cada cliente constituyen un registro para ese cliente. El programa utiliza el número de cuenta como la clave de registro; el archivo se creará y mantendrá en orden basado en el número de cuenta. [*Nota:* Este programa supone que el usuario introduce los registros en orden de número de cuenta. En un sistema comprensivo de cuentas por cobrar, se proporcionaría una herramienta para ordenar datos, de manera que el usuario pudiera introducir los registros en cualquier orden; entonces los registros se ordenarían y se escribirían en el archivo.]

La mayoría de los programas en este capítulo tienen una GUI similar a la que se utiliza en este programa, por lo cual declararemos aquí la clase `IUBanco` (figura 17.5) para encapsular la GUI. (Vea la primera y la tercera capturas de pantalla de la figura 17.7.) Además, el programa declara la clase `RegistroCuentas` (figura 17.6) para encapsular la información de registro del cliente (es decir, cuenta, primer nombre, etc.) utilizada por los ejemplos en este capítulo. Para reutilizarse, las clases `IUBanco` y `RegistroCuenta` se declaran en el paquete `com.deitel.cpej5.cap17`.

Cuando compile las clases `IUBanco` y `RegistroCuenta` o cualesquiera otras clases que se vayan a reutilizar en este capítulo, deberá colocarlas en un directorio común (por ejemplo, `c:\ejemplos\cap17`). Cuando compile o ejecute clases que utilicen a `IUBanco` y `RegistroCuentas`, asegúrese de especificar el argumento de línea de comandos `-classpath` tanto para `javac` como para `java`, como se muestra a continuación:

```
javac -classpath .;c:\ejemplos\cap17 CrearArchivoSecuencial.java
java -classpath .;c:\ejemplos\cap17 CrearArchivoSecuencial
```

y las clases del paquete `com.deitel.cpej5.cap17` se compilan con:

```
javac -d c:\ejemplos\cap17 com\deitel\cpej5\cap17\IUBanco.java
javac -d c:\ejemplos\cap17 com\deitel\cpej5\cap17\RegistroCuentas.java
```

de manera que sus archivos de clase residan en una estructura de directorio que empiece en `c:\ejemplos\cap17`. Asegúrese de incluir el directorio actual (especificado mediante `.`) en la ruta de las clases. Además, el separador de ruta debe ser el apropiado para su plataforma; por ejemplo, un punto y coma (`;`) en Windows y dos puntos (`:`) en UNIX/Linux/Mac OS X.

```
1  // Fig. 17.5: IUBanco.java
2  // Una GUI reutilizable para los ejemplos de este capítulo.
3  package com.deitel.cpej5.cap17;
4
5  import java.awt.*;
6  import javax.swing.*;
7
8  public class IUBanco extends JPanel {
9
10    // texto de las etiquetas para la GUI
11    protected final static String nombres[] = { "Número de cuenta",
12       "Primer nombre", "Apellido", "Saldo", "Monto de la transacción" };
13
14    // componentes de la GUI; protegidos para el acceso futuro de las subclases
15    protected JLabel etiquetas[];
16    protected JTextField campos[];
17    protected JButton hacerTarea1, hacerTarea2;
18    protected JPanel panelInternoCentro, panelInternoSur;
19
20    protected int tamanio; // número de campos de texto en la GUI
21
22    // constantes que representan a los campos de texto en la GUI
23    public static final int CUENTA = 0, PRIMERNOMBRE = 1, APELLIDO = 2,
24       SALDO = 3, TRANSACCION = 4;
```

Figura 17.5 La clase `IUBanco` contiene una GUI reutilizable para varios programas. (Parte 1 de 3.)

http://librosysolucionarios.net

```
25
26    // Configurar la GUI. El argumento miTamanio del constructor determina el
27    // número de filas de componentes de la GUI.
28    public IUBanco( int miTamanio )
29    {
30       tamanio = miTamanio;
31       etiquetas = new JLabel[ tamanio ];
32       campos = new JTextField[ tamanio ];
33
34       // crear etiquetas
35       for ( int cuenta = 0; cuenta < etiquetas.length; cuenta++ )
36          etiquetas[ cuenta ] = new JLabel( nombres[ cuenta ] );
37
38       // crear campos de texto
39       for ( int cuenta = 0; cuenta < campos.length; cuenta++ )
40          campos[ cuenta ] = new JTextField();
41
42       // crear panel para distribuir etiquetas y campos
43       panelInternoCentro = new JPanel();
44       panelInternoCentro.setLayout( new GridLayout( tamanio, 2 ) );
45
46       // adjuntar etiquetas y campos a panelInternoCentro
47       for ( int cuenta = 0; cuenta < tamanio; cuenta++ ) {
48          panelInternoCentro.add( etiquetas[ cuenta ] );
49          panelInternoCentro.add( campos[ cuenta ] );
50       }
51
52       // crear botones genéricos; sin etiquetas ni manejadores de eventos
53       hacerTarea1 = new JButton();
54       hacerTarea2 = new JButton();
55
56       // crear panel para distribuir los botones y adjuntarlos
57       panelInternoSur = new JPanel();
58       panelInternoSur.add( hacerTarea1 );
59       panelInternoSur.add( hacerTarea2 );
60
61       // establecer esquema de este contenedor y adjuntarle los paneles
62       setLayout( new BorderLayout() );
63       add( panelInternoCentro, BorderLayout.CENTER );
64       add( panelInternoSur, BorderLayout.SOUTH );
65
66       validate(); // validar esquema
67
68    } // fin del constructor
69
70    // devolver referencia al botón de tarea genérico hacerTarea1
71    public JButton obtenerBotonHacerTarea1()
72    {
73       return hacerTarea1;
74    }
75
76    // devolver referencia al botón de tarea genérico hacerTarea2
77    public JButton obtenerBotonHacerTarea2()
78    {
79       return hacerTarea2;
```

Figura 17.5 La clase `IUBanco` contiene una GUI reutilizable para varios programas. (Parte 2 de 3.)


```
80    }
81
82    // devolver referencia al arreglo campos de objetos JTextField
83    public JTextField[] obtenerCampos()
84    {
85       return campos;
86    }
87
88    // borrar el contenido de los campos de texto
89    public void borrarCampos()
90    {
91       for ( int cuenta = 0; cuenta < tamanio; cuenta++ )
92          campos[ cuenta ].setText( "" );
93    }
94
95    // establecer valores de los campos de texto; lanzar IllegalArgumentException si
96    // hay un número incorrecto de objetos String en el argumento
97    public void establecerValoresCampos( String cadenas[] )
98       throws IllegalArgumentException
99    {
100      if ( cadenas.length != tamanio )
101         throw new IllegalArgumentException( "Debe haber " +
102            tamanio + " objetos String en el arreglo" );
103
104      for ( int cuenta = 0; cuenta < tamanio; cuenta++ )
105         campos[ cuenta ].setText( cadenas[ cuenta ] );
106   }
107
108   // obtener arreglo de objetos String con el contenido actual de los campos de texto
109   public String[] obtenerValoresCampos()
110   {
111      String valores[] = new String[ tamanio ];
112
113      for ( int cuenta = 0; cuenta < tamanio; cuenta++ )
114         valores[ cuenta ] = campos[ cuenta ].getText();
115
116      return valores;
117   }
118
119 } // fin de la clase IUBanco
```

Figura 17.5 La clase `IUBanco` contiene una GUI reutilizable para varios programas. (Parte 3 de 3.)

La clase `IUBanco` (figura 17.5) contiene dos objetos `JButton` y arreglos de objetos `JLabel` y `JTextField`. El número de objetos `JLabel` y `JTextField` se establece mediante el constructor declarado en las líneas 28 a 68. Los métodos `obtenerValoresCampos` (líneas 109 a 117), `establecerValoresCampos` (líneas 97 a 106) y `borrarCampos` (líneas 89 a 93) manipulan el texto de los objetos `JTextField`. Los métodos `obtenerCampos` (líneas 83 a 86), `obtenerBotonHacerTarea1` (líneas 71 a 74) y `obtenerBotonHacerTarea2` (líneas 77 a 80) devuelve componentes de la GUI individuales para que (por ejemplo) un programa cliente pueda agregar objetos `ActionListener`.

Definición de la clase serializable `RegistroCuentas`

La clase `RegistroCuentas` (figura 17.6) implementa a la interfaz `Serializable`, la cual permite que los objetos de `RegistroCuentas` se utilicen con flujos `ObjectInputStream` y `ObjectOutputStream`. Un flujo `ObjectOutputStream` permite que el objeto se *serialice*; es decir, que se convierta en una serie

http://librosysolucionarios.net

(flujo) de bytes. De manera similar, un flujo **ObjectOutputStream** permite que los datos se *deserialicen*; es decir, que se convierta en una serie (flujo) de bytes. La interfaz **Serializable** es una *interfaz de marcado*. Dicha interfaz no contiene métodos. Una clase que implementa a esta interfaz se *marca* como un objeto **Serializable**, lo cual es importante ya que un flujo **ObjectOutputStream** no enviará un objeto como salida a menos que *sea un* objeto **Serializable**. En una clase que implemente a **Serializable**, el programador debe asegurar que toda variable de instancia de la clase es de un tipo **Serializable**, o debe declarar ciertas variables de instancia como ***transient*** para indicar que esas variables no son **Serializable** y que deben ignorarse durante el proceso de serialización. De manera predeterminada, todas las variables de tipo primitivo son serializables. Para las variables de tipos de referencias, debe comprobar la definición de la clase (y posiblemente sus superclases) para asegurar que el tipo sea **Serializable**. La clase **RegistroCuentas** contiene los miembros de datos **private cuenta**, **primerNombre**, **apellidoPaterno** y **saldo**. Esta clase también proporciona métodos *establecer* y *obtener* **public** para acceder a los campos **private**.

```
1  // Fig. 17.6: RegistroCuentas.java
2  // Una clase que representa un registro de información.
3  package com.deitel.cpej5.cap17;
4
5  import java.io.Serializable;
6
7  public class RegistroCuentas implements Serializable {
8     private int cuenta;
9     private String primerNombre;
10    private String apellidoPaterno;
11    private double saldo;
12
13    // el constructor sin argumentos llama al otro constructor con valores predeterminados
14    public RegistroCuentas()
15    {
16       this( 0, "", "", 0.0 );
17    }
18
19    // inicializar un registro
20    public RegistroCuentas( int cta, String nombre, String apellido, double sald )
21    {
22       establecerCuenta( cta );
23       establecerPrimerNombre( nombre );
24       establecerApellidoPaterno( apellido );
25       establecerSaldo( sald );
26    }
27
28    // establecer número de cuenta
29    public void establecerCuenta( int cta )
30    {
31       cuenta = cta;
32    }
33
34    // obtener número de cuenta
35    public int obtenerCuenta()
36    {
37       return cuenta;
38    }
39
```

Figura 17.6 La clase **RegistroCuentas** mantiene la información para una cuenta. (Parte 1 de 2.)


```
40    // establecer primer nombre
41    public void establecerPrimerNombre( String nombre )
42    {
43       primerNombre = nombre;
44    }
45
46    // obtener primer nombre
47    public String obtenerPrimerNombre()
48    {
49       return primerNombre;
50    }
51
52    // establecer apellido paterno
53    public void establecerApellidoPaterno( String apellido )
54    {
55       apellidoPaterno = apellido;
56    }
57
58     // obtener apellido paterno
59    public String obtenerApellidoPaterno()
60    {
61       return apellidoPaterno;
62    }
63
64    // establecer saldo
65    public void establecerSaldo( double sald )
66    {
67       saldo = sald;
68    }
69
70    // obtener saldo
71    public double obtenerSaldo()
72    {
73       return saldo;
74    }
75
76 } // fin de la clase RegistroCuentas
```

Figura 17.6 La clase **RegistroCuentas** mantiene la información para una cuenta. (Parte 2 de 2.)

Ahora hablemos sobre el código para crear el archivo de acceso secuencial (figura 17.7). En este ejemplo presentamos la clase ***JFileChooser*** (paquete **javax.swing**) para seleccionar archivos (como se muestra en la segunda ventana de la salida). En la línea 94 se crea un objeto **JFileChooser** y se asigna su referencia a **selectorArchivo**. En la línea 95 se hace una llamada al método ***setFileSelectionMode*** para especificar lo que el usuario puede seleccionar de **selectorArchivo**. Para este programa, utilizamos la constante estática ***FILES_ONLY*** de **JFileChooser**, para indicar que sólo pueden seleccionarse archivos. Otras constantes estáticas son ***FILES_AND_DIRECTORIES*** y ***DIRECTORIES_ONLY***.

```
1  // Fig. 17.7: CrearArchivoSecuencial.java
2  // Escribir objetos secuencialmente en un archivo, mediante la clase ObjectOutputStream.
3  import java.io.*;
4  import java.awt.*;
5  import java.awt.event.*;
```

Figura 17.7 Archivo secuencial creado mediante el uso de **ObjectOutputStream**. (Parte 1 de 6.)

http://librosysolucionarios.net

```
6  import javax.swing.*;
7
8  import com.deitel.cpej5.cap17.IUBanco;
9  import com.deitel.cpej5.cap17.RegistroCuentas;
10
11 public class CrearArchivoSecuencial extends JFrame {
12    private ObjectOutputStream salida;
13    private IUBanco interfazUsuario;
14    private JButton botonIntro, botonAbrir;
15
16    // configurar la GUI
17    public CrearArchivoSecuencial()
18    {
19       super( "Creación de un archivo secuencial de objetos" );
20
21       // crear instancia de interfaz de usuario reutilizable
22       interfazUsuario = new IUBanco( 4 ); // cuatro campos de texto
23       getContentPane().add( interfazUsuario, BorderLayout.CENTER );
24
25       // configurar botón hacerTarea1 para usarlo en este programa
26       botonAbrir = interfazUsuario.obtenerBotonHacerTarea1();
27       botonAbrir.setText( "Guardar en archivo ..." );
28
29       // registrar componente de escucha para llamar a abrirArchivo cuando
            se oprima el botón
30       botonAbrir.addActionListener(
31
32          // clase interna anónima para manejar evento de botonAbrir
33          new ActionListener() {
34
35             // llamar a abrirArchivo cuando se oprima el botón
36             public void actionPerformed( ActionEvent evento )
37             {
38                abrirArchivo();
39             }
40
41          } // fin de la clase interna anónima
42
43       ); // fin de la llamada a addActionListener
44
45       // configurar botón hacerTarea2 para usarlo en este programa
46       botonIntro = interfazUsuario.obtenerBotonHacerTarea2();
47       botonIntro.setText( "Introducir" );
48       botonIntro.setEnabled( false );  // deshabilitar botón
49
50       // registrar componente de escucha para llamar a agregarRegistro cuando se
            oprima el botón
51       botonIntro.addActionListener(
52
53          // clase interna anónima para manejar evento de botonIntro
54          new ActionListener() {
55
56             // llamar a agregarRegistro cuando se oprima el botón
57             public void actionPerformed( ActionEvent evento )
58             {
59                agregarRegistro();
60             }
```

Figura 17.7 Archivo secuencial creado mediante el uso de `ObjectOutputStream`. (Parte 2 de 6.)


```
61
62          } // fin de la clase interna anónima
63
64       ); // fin de la llamada a addActionListener
65
66       // registrar componente de escucha de ventana para manejar evento de cierre
            de ventana
67       addWindowListener(
68
69          // clase interna anónima para manejar evento windowClosing
70          new WindowAdapter() {
71
72             // agregar registro actual en la GUI al archivo, después cerrar el
                archivo
73             public void windowClosing( WindowEvent evento )
74             {
75                if ( salida != null )
76                   agregarRegistro();
77
78                cerrarArchivo();
79             }
80
81          } // fin de la clase interna anónima
82
83       ); // fin de la llamada a addWindowListener
84
85       setSize( 300, 200 );
86       setVisible( true );
87
88    } // fin del constructor de CrearArchivoSecuencial
89
90    // permitir al usuario especificar el nombre del archivo
91    private void abrirArchivo()
92    {
93       // mostrar cuadro de diálogo de archivo, para que el usuario pueda elegir el
            archivo a abrir
94       JFileChooser selectorArchivo = new JFileChooser();
95       selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );
96
97       int resultado = selectorArchivo.showSaveDialog( this );
98
99       // si el usuario hizo clic en el botón cancelar del cuadro de diálogo, regresar
100      if ( resultado == JFileChooser.CANCEL_OPTION )
101         return;
102
103      File nombreArchivo = selectorArchivo.getSelectedFile(); // obtener archivo
            seleccionado
104
105      // mostrar error si es inválido
106      if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) )
107         JOptionPane.showMessageDialog( this, "Nombre de archivo inválido",
108            "Nombre de archivo inválido", JOptionPane.ERROR_MESSAGE );
109
110      else {
111
112         // abrir archivo
113         try {
114            salida = new ObjectOutputStream(
```

Figura 17.7 Archivo secuencial creado mediante el uso de `ObjectOutputStream`. (Parte 3 de 6.)

http://librosysolucionarios.net

```
            new FileOutputStream( nombreArchivo ) );

            botonAbrir.setEnabled( false );
            botonIntro.setEnabled( true );
         }

      // procesar excepciones que pueden ocurrir al abrir el archivo
         catch ( IOException excepcionES ) {
            JOptionPane.showMessageDialog( this, "Error al abrir el archivo",
               "Error", JOptionPane.ERROR_MESSAGE );
         }

      } // fin de instrucción else

   } // fin del método abrirArchivo

   // cerrar archivo y terminar la aplicación
   private void cerrarArchivo()
   {
      // cerrar el archivo
      try {
         salida.close();
         System.exit( 0 );
      }

      // procesar excepciones que pueden ocurrir al cerrar el archivo
      catch( IOException excepcionES ) {
         JOptionPane.showMessageDialog( this, "Error al cerrar el archivo",
            "Error", JOptionPane.ERROR_MESSAGE );
         System.exit( 1 );
      }

   } // fin del método cerrarArchivo

   // agregar registro al archivo
   public void agregarRegistro()
   {
      int numeroCuenta = 0;
      RegistroCuentas registro;
      String valoresCampos[] = interfazUsuario.obtenerValoresCampos();

      // si el valor del campo cuenta no está vacío
      if ( ! valoresCampos[ IUBanco.CUENTA ].equals( "" ) ) {

         // escribir valores en el archivo
         try {
            numeroCuenta = Integer.parseInt(
               valoresCampos[ IUBanco.CUENTA ] );

            if ( numeroCuenta > 0 ) {

               // crear nuevo registro
               registro = new RegistroCuentas( numeroCuenta,
                  valoresCampos[ IUBanco.PRIMERNOMBRE ],
                  valoresCampos[ IUBanco.APELLIDO ],
                  Double.parseDouble( valoresCampos[ IUBanco.SALDO ] ) );
```

Figura 17.7 Archivo secuencial creado mediante el uso de `ObjectOutputStream`. (Parte 4 de 6.)


```

               // escribir el registro y vaciar el búfer
               salida.writeObject( registro );
               salida.flush();
            }
            else
            {
               JOptionPane.showMessageDialog( this,
                  "El número de cuenta debe ser mayor que 0",
                  "Número de cuenta incorrecto", JOptionPane.ERROR_MESSAGE );
            }

            // borrar campos de texto
            interfazUsuario.borrarCampos();

         } // fin de bloque try

         // procesar formato inválido de número de cuenta o saldo
         catch ( NumberFormatException excepcionFormato ) {
            JOptionPane.showMessageDialog( this,
               "Número de cuenta o saldo incorrecto", "Formato de número incorrecto",
               JOptionPane.ERROR_MESSAGE );
         }

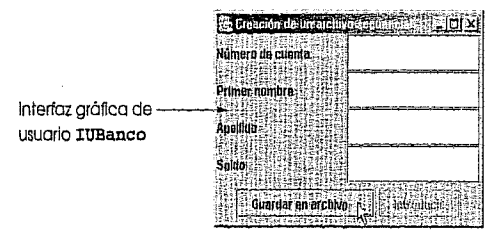
         // procesar excepciones que pueden ocurrir al escribir en el archivo
         catch ( IOException excepcionES ) {
            JOptionPane.showMessageDialog( this, "Error al escribir en el archivo",
               "Excepción de ES", JOptionPane.ERROR_MESSAGE );
            cerrarArchivo();
         }

      } // fin de instrucción if

   } // fin del método agregarRegistro

   public static void main( String args[] )
   {
      new CrearArchivoSecuencial();
   }

} // fin de la clase CrearArchivoSecuencial
```



Interfaz gráfica de usuario **IUBanco**

Figura 17.7 Archivo secuencial creado mediante el uso de `ObjectOutputStream`. (Parte 5 de 6.)

http://librosysolucionarios.net

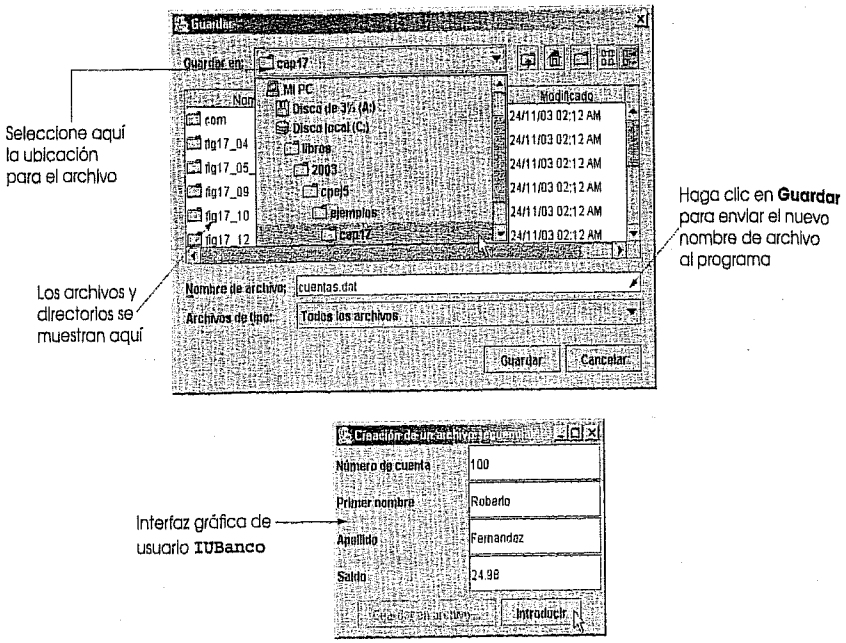


Figura 17.7 Archivo secuencial creado mediante el uso de `ObjectOutputStream`. (Parte 6 de 6.)

En la línea 97 se hace una llamada al método *`showSaveDialog`* para mostrar el cuadro de diálogo de `JFileChooser` titulado **Guardar**. El argumento `this` especifica la ventana *padre* del cuadro de diálogo `JFileChooser`, la cual determina la posición del cuadro de diálogo en la pantalla. Si se pasa `null`, el cuadro de diálogo se muestra en el centro de la pantalla; en caso contrario, el cuadro de diálogo se centra sobre la ventana de aplicación. Un cuadro de diálogo `JFileChooser` es un cuadro de diálogo modal que no permite al usuario interactuar con ninguna otra ventana de programa, sino hasta que cierre el cuadro de diálogo `JFileChooser` haciendo clic en **Guardar** o **Cancelar**. El usuario selecciona la unidad, directorio y nombre el archivo, y después hace clic en **Guardar**. El método `showSaveDialog` devuelve un entero que especifica en cuál botón (**Guardar** o **Cancelar**) hizo clic el usuario para cerrar el cuadro de diálogo. En la línea 100 se evalúa si el usuario hizo clic en **Cancelar**, comparando el valor de `resultado` con la constante `static` *`CANCEL_OPTION`*. Si son iguales, el método regresa.

En la línea 103 se recupera el archivo que seleccionó el usuario, llamando al método *`getSelectedFile`*, el cual devuelve un objeto de tipo `File` que encapsula la información acerca del archivo (por ejemplo, nombre y ubicación), pero no representa el contenido del archivo. Este objeto `File` no abre el archivo. Asignamos la referencia a este objeto `File` a la variable `nombreArchivo`.

Como lo indicamos anteriormente, un programa puede abrir un archivo creando un objeto de la clase de flujo `FileInputStream` o `FileOutputStream`. En este ejemplo, el archivo se va a abrir en modo de salida, por lo que el programa crea un objeto `FileOutputStream` (línea 115). Se pasa un argumento al constructor de `FileOutputStream`: un objeto `File`. Los archivos existentes que se abren en modo de salida de esta forma se *truncan* (se descartan todos los datos en el archivo).



Error común de programación 17.2

Es un error lógico abrir un archivo existente en modo de salida cuando, de hecho, el usuario desea preservar el archivo. Para anexar datos a un archivo, utilice el constructor de `FileOutputStream` *hasta dos argumentos con* `true` *como segundo argumento.*

http://librosysolucionarios.net

La clase `FileOutputStream` proporciona métodos para escribir arreglos `byte` y objetos `byte` individuales en un archivo. Para este programa, escribiremos objetos en un archivo; esta capacidad no la proporciona `FileOutputStream`. Por esta razón envolvemos un objeto `FileOutputStream` en un objeto `ObjectOutputStream`, pasando el nuevo objeto `FileOutputStream` al constructor de `ObjectOutputStream` (líneas 114 y 115). El constructor podría lanzar una excepción *`IOException`* si ocurre un problema al abrir el archivo (por ejemplo, cuando se abre un archivo en modo de escritura, en una unidad de disco con espacio insuficiente o cuando se abre un archivo de sólo lectura en modo de escritura). De ser así, el programa muestra un mensaje de error (líneas 122 a 125). Si la construcción de los dos flujos no lanza una excepción `IOException`, el archivo se abre.

Este programa supone que los datos se introducen correctamente y en el orden de número de registro apropiado. El usuario escribe los datos en objetos `JTextFields` y hace clic en **Introducir** para escribir los datos en el archivo. El método `actionPerformed` del botón **Introducir** (líneas 57 a 60) llama a nuestro método `agregarRegistro` (líneas 150 a 204) para llevar a cabo la operación de escritura. En la línea 173 se hace una llamada al método *`writeObject`* para escribir el objeto `registro` en el archivo de salida. En la línea 174 se hace una llamada al método *`flush`* para asegurar que los datos almacenados en memoria se escriban inmediatamente en el archivo.

Cuando el usuario cierra la ventana de aplicación, el programa llama al método `windowClosing` (líneas 73 a 79), el cual compara a `salida` con `null` (línea 75). Si `salida` no es `null`, el flujo se abre y se hace una llamada a los métodos `agregarRegistro` (líneas 150 a 204) y `cerrarArchivo` (líneas 132 a 147). El método `cerrarArchivo` llama al método *`close`* en `salida` para cerrar el archivo. Observe que la llamada al método `close` está encerrada en un bloque `try`. El método `close` lanza una excepción `IOException` si el archivo no puede cerrarse apropiadamente. En este caso, es importante notificar al usuario que la información en el archivo podría estar corrupta.



Tip de rendimiento 17.2

Siempre libere los recursos explícitamente y en cuanto que se determine que los recursos ya no se necesitan. Este procedimiento hace a los recursos inmediatamente disponibles para ser reutilizados por su programa o por otro programa, mejorando así la utilización de los recursos.

Al usar flujos envueltos, el flujo más externo (`ObjectOutputStream` en este ejemplo) debe utilizarse para cerrar el archivo.



Tip de rendimiento 17.3

Cierre explícitamente cada archivo, tan pronto como se sepa que el programa no hará referencia a él de nuevo. Esta práctica puede reducir el uso de recursos en un programa que seguirá ejecutándose mucho tiempo después de que ya no necesite hacer referencia a un archivo específico. Esta práctica también mejora la claridad del programa.

En la ejecución de ejemplo del programa de la figura 17.7, introducimos información para cinco cuentas. (Vea la figura 17.8.) El programa no muestra cómo aparecen realmente los registros de datos en el archivo. Para verificar que el archivo se haya creado correctamente, en la siguiente sección presentaremos un programa para leer el contenido del archivo.

Datos de ejemplo

100	Roberto	Fernández	24.98
200	Juan	Torres	-345.67
300	María	González	0.00
400	Samuel	Martínez	-42.16
500	Susana	Robles	224.62

Figura 17.8 Datos de ejemplo para el programa de la figura 17.7.

```
1  // Fig. 17.9: LeerArchivoSecuencial.java
2  // Este programa lee un archivo de objetos en forma secuencial
3  // y muestra cada uno de los registros.
4  import java.io.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  import com.deitel.cpej5.cap17.*;
10
11 public class LeerArchivoSecuencial extends JFrame {
12    private ObjectInputStream entrada;
13    private IUBanco interfazUsuario;
14    private JButton botonSiguiente, botonAbrir;
15
16    // Constructor -- inicializar el marco
17    public LeerArchivoSecuencial()
18    {
19       super( "Leer un archivo secuencial de objetos" );
20
21       // crear instancia de la interfaz de usuario reutilizable
22       interfazUsuario = new IUBanco( 4 );  // cuatro campos de texto
23       getContentPane().add( interfazUsuario, BorderLayout.CENTER );
24
25       // obtener referencia al botón de tarea genérico hacerTarea1 de IUBanco
26       botonAbrir = interfazUsuario.obtenerBotonHacerTarea1();
27       botonAbrir.setText( "Abrir archivo" );
28
29       // registrar componente de escucha para llamar a abrirArchivo cuando se
            oprima el botón
30       botonAbrir.addActionListener(
31
32          // clase interna anónima para manejar evento de botonAbrir
33          new ActionListener() {
34
35             // cerrar archivo y terminar la aplicación
36             public void actionPerformed( ActionEvent evento )
37             {
38                abrirArchivo();
39             }
40
41          } // fin de la clase interna anónima
42
43       ); // fin de la llamada a addActionListener
44
45       // registrar componente de escucha de ventana para evento de cierre de ventana
46       addWindowListener(
47
48          // clase interna anónima para manejar evento windowClosing
49          new WindowAdapter() {
50
51             // cerrar el archivo y terminar la aplicación
52             public void windowClosing( WindowEvent evento )
53             {
54                if ( entrada != null )
55                   cerrarArchivo();
```

Figura 17.9 Lectura de un archivo secuencial, utilizando `ObjectInputStream`. (Parte 1 de 5.)


```
56
57                System.exit( 0 );
58             }
59
60          } // fin de la clase interna anónima
61
62       ); // fin de la llamada a addWindowListener
63
64       // obtener referencia al botón de tarea genérico hacerTarea2 de IUBanco
65       botonSiguiente = interfazUsuario.obtenerBotonHacerTarea2();
66       botonSiguiente.setText( "Siguiente registro" );
67       botonSiguiente.setEnabled( false );
68
69       // registrar componente de escucha para llamar a leerRegistro cuando se
            oprima el botón
70       botonSiguiente.addActionListener(
71
72          // clase interna anónima para manejar evento de siguienteRegistro
73          new ActionListener() {
74
75             // llamar a leerRegistro cuando el usuario haga clic en siguienteRegistro
76             public void actionPerformed( ActionEvent evento )
77             {
78                leerRegistro();
79             }
80
81          } // fin de la clase interna anónima
82
83       ); // fin de la llamada a addActionListener
84
85       pack();
86       setSize( 300, 200 );
87       setVisible( true );
88
89    } // fin del constructor de LeerArchivoSecuencial
90
91    // permitir al usuario seleccionar el archivo a abrir
92    private void abrirArchivo()
93    {
94       // mostrar el cuadro de diálogo del archivo, para que el usuario pueda
            seleccionar el archivo a abrir
95       JFileChooser selectorArchivo = new JFileChooser();
96       selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );
97
98       int resultado = selectorArchivo.showOpenDialog( this );
99
100      // si el usuario hizo clic en el botón Cancelar en el cuadro de diálogo, regresar
101      if ( resultado == JFileChooser.CANCEL_OPTION )
102         return;
103
104      // obtener el archivo seleccionado
105      File nombreArchivo = selectorArchivo.getSelectedFile();
106
107      // mostrar error si el nombre de archivo es incorrecto
108      if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) )
109         JOptionPane.showMessageDialog( this, "Nombre de archivo incorrecto",
110            "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
```

http://librosysolucionarios.net

Figura 17.9 Lectura de un archivo secuencial, utilizando `ObjectInputStream`. (Parte 2 de 5.)

```
111
112        else {
113
114           // abrir archivo
115           try {
116              entrada = new ObjectInputStream(
117                 new FileInputStream( nombreArchivo ) );
118
119              botonAbrir.setEnabled( false );
120              botonSiguiente.setEnabled( true );
121           }
122
123           // procesar excepciones que pueden ocurrir al abrir el archivo
124           catch ( IOException excepcionES ) {
125              JOptionPane.showMessageDialog( this, "Error al abrir el archivo",
126                 "Error", JOptionPane.ERROR_MESSAGE );
127           }
128
129        } // fin de instrucción else
130
131     } // fin del método abrirArchivo
132
133     // leer registro del archivo
134     public void leerRegistro()
135     {
136        RegistroCuentas registro;
137
138        // leer los valores del archivo
139        try {
140           registro = ( RegistroCuentas ) entrada.readObject();
141
142           // crear arreglo de objetos String a mostrar en la GUI
143           String valores[] = { String.valueOf( registro.obtenerCuenta() ),
144              registro.obtenerPrimerNombre(), registro.obtenerApellidoPaterno(),
145              String.valueOf( registro.obtenerSaldo() ) };
146
147           // mostrar contenido del registro
148           interfazUsuario.establecerValoresCampos( valores );
149        }
150
151        // mostrar mensaje al llegar al fin del archivo
152        catch ( EOFException excepcionFinDeArchivo ) {
153           botonSiguiente.setEnabled( false );
154
155           JOptionPane.showMessageDialog( this, "No hay más registros en el archivo",
156              "Fín del archivo", JOptionPane.ERROR_MESSAGE );
157        }
158
159        // mostrar mensaje de error si no se encuentra la clase
160        catch ( ClassNotFoundException excepcionClaseNoEncontrada ) {
161           JOptionPane.showMessageDialog( this, "No se pudo crear el objeto",
162              "Clase no encontrada", JOptionPane.ERROR_MESSAGE );
163        }
164
165        // mostrar mensaje de error si no se puede leer debido a un problema con el archivo
```

Figura 17.9 Lectura de un archivo secuencial, utilizando `ObjectInputStream`. (Parte 3 de 5.)


```
166        catch ( IOException excepcionES ) {
167           JOptionPane.showMessageDialog( this,
168              "Error al leer del archivo",
169              "Error de lectura", JOptionPane.ERROR_MESSAGE );
170        }
171
172     } // fin del método leerRegistro
173
174     // cerrar archivo y terminar la aplicación
175     private void cerrarArchivo()
176     {
177        // cerrar archivo y salir
178        try {
179           entrada.close();
180           System.exit( 0 );
181        }
182
183        // procesar excepción que puede ocurrir mientras se cierra el archivo
184        catch ( IOException excepcionES ) {
185           JOptionPane.showMessageDialog( this, "Error al cerrar el archivo",
186              "Error", JOptionPane.ERROR_MESSAGE );
187
188           System.exit( 1 );
189        }
190
191     } // fin del método cerrarArchivo
192
193     public static void main( String args[] )
194     {
195        new LeerArchivoSecuencial();
196     }
197
198  } // fin de la clase LeerArchivoSecuencial
```

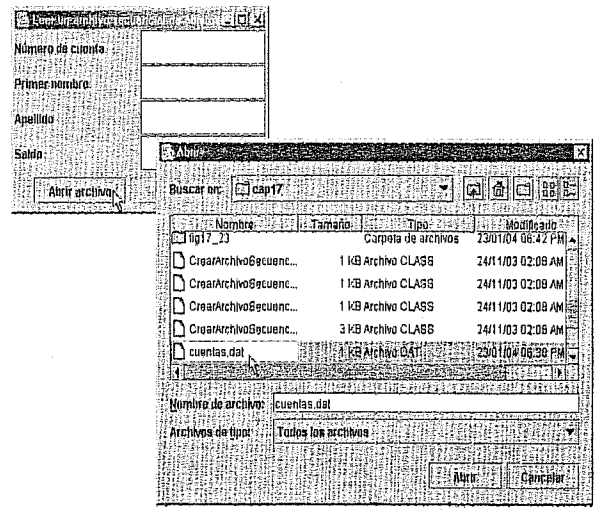


Figura 17.9 Lectura de un archivo secuencial, utilizando `ObjectInputStream`. (Parte 4 de 5.)

http://librosysolucionarios.net

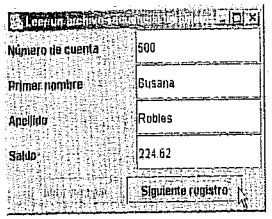


Figura 17.9 Lectura de un archivo secuencial, utilizando `ObjectInputStream`. (Parte 5 de 5.)

17.6 Cómo leer datos desde un archivo de acceso secuencial

Los datos se almacenan en archivos, para que puedan recuperarse y procesarse según sea necesario. En la sección anterior demostramos cómo crear un archivo con acceso secuencial. En esta sección veremos cómo leer datos de un archivo en forma secuencial.

El programa de la figura 17.9 lee los registros de un archivo creado por el programa de la figura 17.7 y muestra el contenido de esos registros. El programa abre el archivo en modo de entrada, creando un objeto `FileInputStream`. El programa especifica el nombre del archivo a abrir como un argumento para el constructor de `FileInputStream`. En la figura 17.7 escribimos objetos en el archivo, utilizando un objeto `ObjectOutputStream`. Los datos deben leerse desde el archivo, en el mismo formato en el que se escribieron en el archivo. Por lo tanto, en este programa utilizamos un objeto `ObjectInputStream` envuelto alrededor de un objeto `FileInputStream`. Observe que la tercera captura de pantalla de ejemplo muestra a la GUI con el último registro en el archivo.

La mayor parte del código en este ejemplo es similar al de la figura 17.7, por lo que sólo hablaremos de las líneas de código clave que son distintas. En la línea 98 se hace una llamada al método ***showOpenDialog*** de `JFileChooser` para mostrar el cuadro de diálogo **Abrir** (la segunda captura de pantalla de la figura 17.9). El comportamiento y la GUI son iguales que en el cuadro de diálogo mostrado por `showSaveDialog`, con la excepción de que el título del cuadro de diálogo y el botón **Guardar** se sustituyen con **Abrir**.

En las líneas 116 y 117 se crea un objeto `ObjectInputStream` y se asigna su referencia a `input`. El objeto `File` que contiene el nombre de archivo que seleccionó el usuario se pasa al constructor de `FileInputStream` para abrir el archivo.

El programa lee un registro del archivo cada vez que el usuario hace clic en **Siguiente registro**. En la línea 78, en el método `actionPerformed` del botón **Siguiente registro** se hace una llamada al método `leerRegistro` (líneas 134 a 172) para leer un registro del archivo. En la línea 140 se hace una llamada al método ***readObject*** para leer un objeto `Object` del flujo `ObjectInputStream`. Para utilizar métodos específicos de `RegistroCuentas`, convertimos el objeto `Object` devuelto al tipo `RegistroCuentas`. Si se llega al marcador de fin de archivo durante la lectura, `readObject` lanza una excepción ***EndOfFileException***.

Programa para consultar créditos

Para recuperar datos en forma secuencial de un archivo, los programas generalmente empiezan a leer desde el principio del archivo y van leyendo todos los datos consecutivamente, hasta que se encuentra la información deseada. Podría ser necesario procesar el archivo en forma secuencial varias veces (desde el principio del archivo) durante la ejecución de un programa. La clase `FileInputStream` no proporciona la habilidad de reposicionarse en el principio del archivo para leerlo otra vez, a menos que el programa cierre el archivo y vuelva a abrirlo. Los objetos `RandomAccessFile` pueden reposicionarse en el principio del archivo. La clase `RandomAccessFile` (secciones 17.8 a 17.12) proporciona todas las herramientas de las clases `FileInputStream`, `FileOutputStream`, `DataInputStream` y `DataOutputStream`, además de agregar varios métodos más, incluyendo un método ***seek*** que reposiciona el *apuntador de posición de archivo* (el número de byte del siguiente byte en el archivo que se va a leer o en el que se va a escribir) en cualquier posición del archivo. Sin embargo, la clase `RandomAccessFile` no puede leer y escribir objetos completos. Es por esta razón que necesitamos cerrar y volver a abrir el archivo, para permitir a nuestro programa leer datos desde el principio del archivo varias veces.

Tip de rendimiento 17.4



El proceso de cerrar y volver a abrir un archivo con el fin de posicionar el apuntador de posición de archivo de vuelta en el principio del archivo es una tarea en la que la computadora consume mucho tiempo. Si esta operación se hace con frecuencia, puede reducir el rendimiento de su programa. Si su programa necesita que se utilice frecuentemente el reposicionamiento antes descrito, considere el uso de un archivo de acceso aleatorio.

El programa de la figura 17.10 permite a un gerente de créditos mostrar la información de la cuenta de aquellos clientes con saldos de cero (es decir, los clientes que no deben dinero a la compañía), saldos con crédito (es decir, los clientes a quienes la compañía les debe dinero) y saldos con débito (es decir, los clientes que deben dinero a la compañía por los bienes y servicios recibidos en el pasado). El programa muestra botones que permiten a un gerente de créditos obtener la información crediticia. El botón **Saldos con crédito** produce una lista de cuentas con saldos con crédito. El botón **Saldos con débito** produce una lista de cuentas con saldos con débito. El botón **Saldos en ceros** produce una lista de cuentas con saldos en ceros.

Los registros se muestran en un objeto `JTextArea` llamada `areaMostrarRegistros`. La información de los registros se recolecta leyendo a través del archivo completo y determinando, para cada registro, si cumple o no con los criterios para el tipo de cuenta seleccionado por el gerente de créditos. Al hacer clic en uno de los botones de saldos se asigna a la variable `tipoCuenta` (línea 242) el texto del botón que fue oprimido (por ejemplo, **Saldos en ceros**) y se invoca al método `leerRegistros` (líneas 167 a 214), el cual itera a través del archivo y lee cada uno de los registros. En la línea 189 del método `leerRegistros` se hace una llamada al método `debeMostrarse` (líneas 217 a 229) para determinar si el registro actual cumple con el tipo de cuenta solicitado. Si `debeMostrarse` devuelve `true`, el programa anexa la información de cuenta del registro actual al área de texto `mostrarRegistros`. Cuando se llega al marcador de fin de archivo, en la línea 199 se hace una llamada al método `cerrarArchivo` para cerrar el archivo.

```
1  // Fig. 17.10: ConsultaCreditos.java
2  // Este programa lee un archivo en forma secuencial y muestra su contenido en un
3  // área de texto, con base en el tipo de cuenta que el usuario solicita
4  // (saldo con crédito, saldo con débito o saldo en ceros).
5  import java.io.*;
6  import java.awt.*;
7  import java.awt.event.*;
8  import java.text.DecimalFormat;
9  import javax.swing.*;
10
11 import com.deitel.cpei5.cap17.RegistroCuentas;
12
13 public class ConsultaCreditos extends JFrame {
14    private JTextArea areaMostrarRegistros;
15    private JButton botonAbrir, botonCredito, botonDebito, botonCeros;
16    private JPanel panelBotones;
17
18    private ObjectInputStream entrada;
19    private FileInputStream entradaArchivo;
20    private File nombreArchivo;
21    private String tipoCuenta;
22
23    static private DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
24
25    // configurar la GUI
26    public ConsultaCreditos()
```

Figura 17.10 Programa para consulta de créditos. (Parte 1 de 6.)

http://librosysolucionarios.net

```
27    {
28       super( "Programa de consulta de créditos" );
29
30       Container contenedor = getContentPane();
31
32       panelBotones = new JPanel(); // configurar panel para agregarle botones
33
34       // crear y configurar botón para abrir el archivos
35       botonAbrir = new JButton( "Abrir archivo" );
36       panelBotones.add( botonAbrir );
37
38       // registrar componente de escucha de botonAbrir
39       botonAbrir.addActionListener(
40
41          // clase interna anónima para manejar evento de botonAbrir
42          new ActionListener() {
43
44             // abrir archivo para procesarlo
45             public void actionPerformed( ActionEvent evento )
46             {
47                abrirArchivo();
48             }
49
50          } // fin de la clase interna anónima
51
52       ); // fin de la llamada a addActionListener
53
54       // crear y configurar botón para obtener cuentas con saldos con crédito
55       botonCredito = new JButton( "Saldos con crédito" );
56       panelBotones.add( botonCredito );
57       botonCredito.addActionListener( new ManejadorBotones() );
58
59       // crear y configurar botón para obtener cuentas con saldos con débito
60       botonDebito = new JButton( "Saldos con débito" );
61       panelBotones.add( botonDebito );
62       botonDebito.addActionListener( new ManejadorBotones() );
63
64       // crear y configurar botón para obtener cuentas con saldos en ceros
65       botonCeros = new JButton( "Saldos en ceros" );
66       panelBotones.add( botonCeros );
67       botonCeros.addActionListener( new ManejadorBotones() );
68
69       // establecer área para mostrar resultados
70       areaMostrarRegistros = new JTextArea();
71       JScrollPane desplazador = new JScrollPane( areaMostrarRegistros );
72
73       // adjuntar componentes al panel de contenido
74       contenedor.add( desplazador, BorderLayout.CENTER );
75       contenedor.add( panelBotones, BorderLayout.SOUTH );
76
77       botonCredito.setEnabled( false ); // deshabilitar botonCredito
78       botonDebito.setEnabled( false );  // deshabilitar botonDebito
79       botonCeros.setEnabled( false );   // deshabilitar botonCeros
80
81       // registrar componente de escucha de ventana
```

Figura 17.10 Programa para consulta de créditos. (Parte 2 de 6.)


```
82       addWindowListener(
83
84          // clase interna anónima para evento windowClosing
85          new WindowAdapter() {
86
87             // cerrar archivo y terminar el programa
88             public void windowClosing( WindowEvent evento )
89             {
90                cerrarArchivo();
91                System.exit( 0 );
92             }
93
94          } // fin de la clase interna anónima
95
96       ); // fin de la llamada a addWindowListener
97
98       pack(); // empaquetar componentes y mostrar ventana
99       setSize( 600, 250 );
100       setVisible( true );
101
102    } // fin del constructor de ConsultaCreditos
103
104    // permitir al usuario seleccionar el archivo a abrir
105    private void abrirArchivo()
106    {
107       // mostrar cuadro de diálogo, para que el usuario pueda seleccionar el archivo
108       JFileChooser selectorArchivo = new JFileChooser();
109       selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );
110
111       int resultado = selectorArchivo.showOpenDialog( this );
112
113       // si el usuario hizo clic en el botón Cancelar del cuadro de diálogo, regresar
114       if ( resultado == JFileChooser.CANCEL_OPTION )
115          return;
116
117       nombreArchivo = selectorArchivo.getSelectedFile(); // obtener archivo
          seleccionado
118
119       // mostrar error si el nombre de archivo es incorrecto
120       if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) )
121          JOptionPane.showMessageDialog( this, "Nombre de archivo incorrecto",
122             "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
123
124       // abrir el archivo
125       try {
126
127          // cerrar archivo de la operación anterior
128          if ( entrada != null )
129             entrada.close();
130
131          entradaArchivo = new FileInputStream( nombreArchivo );
132          entrada = new ObjectInputStream( entradaArchivo );
133          botonAbrir.setEnabled( false );
134          botonCredito.setEnabled( true );
135          botonDebito.setEnabled( true );
136          botonCeros.setEnabled( true );
```

http://librosysolucionarios.net

Figura 17.10 Programa para consulta de créditos. (Parte 3 de 6.)

```
137         }
138
139         // atrapar problemas que pueden ocurrir al manipular el archivo
140         catch ( IOException excepcionES ) {
141            JOptionPane.showMessageDialog( this, "El archivo no existe",
142               "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
143         }
144
145      } // fin del método abrirArchivo
146
147      // cerrar archivo antes de que termine la aplicación
148      private void cerrarArchivo()
149      {
150         // cerrar el archivo
151         try {
152            if ( entrada != null )
153               entrada.close();
154         }
155
156         // procesar excepciones que puedan ocurrir al cerrar el archivo
157         catch ( IOException excepcionES ) {
158            JOptionPane.showMessageDialog( this, "Error al cerrar el archivo",
159               "Error", JOptionPane.ERROR_MESSAGE );
160
161            System.exit( 1 );
162         }
163
164      } // fin del método cerrarArchivo
165
166      // leer registros del archivo y mostrar sólo los registros del tipo apropiado
167      private void leerRegistros()
168      {
169         RegistroCuentas registro;
170
171         // leer registros
172         try {
173
174            if ( entrada != null )
175               entrada.close();
176
177            entradaArchivo = new FileInputStream( nombreArchivo );
178            entrada = new ObjectInputStream( entradaArchivo );
179
180            areaMostrarRegistros.setText( "Las cuentas son:\n" );
181
182            // recibir como entrada los valores del archivo
183            while ( true ) {
184
185               // leer un RegistroCuentas
186               registro = ( RegistroCuentas ) entrada.readObject();
187
188               // si es el tipo de cuenta apropiado, mostrar el registro
189               if ( debeMostrarse( registro.obtenerSaldo() ) )
190                  areaMostrarRegistros.append( registro.obtenerCuenta
191                     () + "\t" + registro.obtenerPrimerNombre() + "\t" +
                        registro.obtenerApellidoPaterno() +
```

Figura 17.10 Programa para consulta de créditos. (Parte 4 de 6.)


```
192                  "\t" + dosDigitos.format( registro.obtenerSaldo() ) + "\n" );
193            }
194
195         } // fin del bloque try
196
197         // cerrar archivo cuando se llega al fin de archivo
198         catch ( EOFException excepcionEOF ) {
199            cerrarArchivo();
200         }
201
202         // mostrar error si no se puede leer el objeto por no encontrar la clase
203         catch ( ClassNotFoundException claseNoEncontrada ) {
204            JOptionPane.showMessageDialog( this, "No se pudo crear el objeto",
205               "Clase no encontrada", JOptionPane.ERROR_MESSAGE );
206         }
207
208         // mostrar error si no se puede leer debido a un problema con el archivo
209         catch ( IOException excepcionES ) {
210            JOptionPane.showMessageDialog( this, "Error al leer del archivo",
211               "Error", JOptionPane.ERROR_MESSAGE );
212         }
213
214      } // fin del método leerRegistros
215
216      // use el tipo del registro para determinar si el registro debe mostrarse
217      private boolean debeMostrarse( double saldo )
218      {
219         if ( tipoCuenta.equals( "Saldos con crédito" ) && saldo < 0 )
220            return true;
221
222         else if ( tipoCuenta.equals( "Saldos con débito" ) && saldo > 0 )
223            return true;
224
225         else if ( tipoCuenta.equals( "Saldos en ceros" ) && saldo == 0 )
226            return true;
227
228         return false;
229      }
230
231      public static void main( String args[] )
232      {
233         new ConsultaCreditos();
234      }
235
236      // clase para el manejo de eventos de botonCredito, botonDebito y botonCeros
237      private class ManejadorBotones implements ActionListener {
238
239         // leer registros del archivo
240         public void actionPerformed( ActionEvent evento )
241         {
242            tipoCuenta = evento.getActionCommand();
243            leerRegistros();
244         }
245
246      } // fin de la clase ManejadorBotones
```

http://librosysolucionarios.net

Figura 17.10 Programa para consulta de créditos. (Parte 5 de 6.)

```
247
248 } // fin de la clase ConsultaCreditos
```

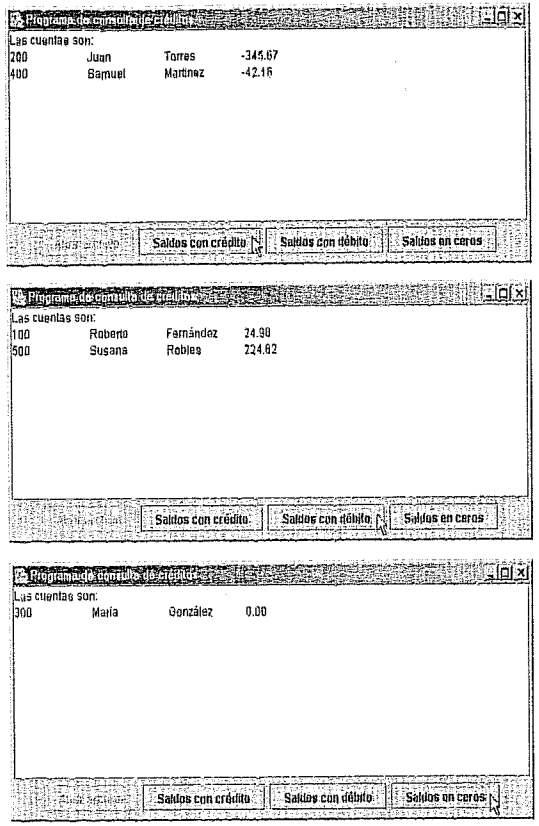


Figura 17.10 Programa para consulta de créditos. (Parte 6 de 6.)

17.7 Actualización de archivos de acceso secuencial

Los datos a los que se les da formato y se escriben en un archivo de acceso secuencial, como se muestra en la sección 17.5, no pueden modificarse sin tener que leer y escribir todos los datos en el archivo. Por ejemplo, si el nombre `Villarreal` necesita cambiarse por `Villanueva`, el nombre anterior no puede simplemente sobrescribirse. Dicha actualización puede llevarse a cabo, pero es un proceso complicado. Para hacer que el nombre anterior cambie, los registros que están antes de `Villarreal` en un archivo de acceso secuencial se copian a un nuevo archivo. Después el registro actualizado se escribe al nuevo archivo, y los registros que están después de `Villarreal` se copian al nuevo archivo. Esta operación requiere de procesar todos los registros en el archivo, para poder actualizar un registro. Si se van a actualizar muchos registros en una pasada del archivo, esta técnica puede ser aceptable.

17.8 Archivos de acceso aleatorio

Hasta ahora hemos visto cómo crear archivos de acceso secuencial, y cómo realizar búsquedas en ellos para localizar cierta información. Los archivos de acceso secuencial son inapropiados para las denominadas *aplicaciones de acceso instantáneo*, en las cuales un registro específico de información debe localizarse inmediatamente. Algunas aplicaciones populares de acceso instantáneo son los sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta, máquinas de cajeros automáticos y demás tipos de *sistemas de procesamiento de transacciones* que requieren de un rápido acceso a los datos específicos. El banco en el que usted tiene su cuenta podría tener cientos de miles, o incluso millones, de clientes; y aún así, cuando usted utiliza una máquina de cajero automático, el banco determina en segundos si su cuenta tiene fondos suficientes para la transacción. Este tipo de acceso instantáneo es posible mediante los *archivos de acceso aleatorio* y las bases de datos (capítulo 23). Un programa puede acceder a los registros individuales de un archivo de acceso aleatorio directa y rápidamente, sin tener que buscar a través de los demás registros. A los archivos de acceso aleatorio se les conoce algunas veces como *archivos de acceso directo*.

http://librosysolucionarios.net

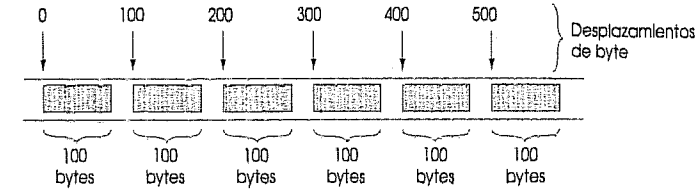


Figura 17.11 La manera en que Java ve a un archivo de acceso aleatorio.

Recuerde que en la sección 17.5 vimos que Java no impone la estructura en un archivo, por lo cual una aplicación que desee utilizar archivos de acceso aleatorio deberá especificar el formato de esos archivos. Pueden usarse varias técnicas para crear archivos de acceso aleatorio. Tal vez la más simple sea requerir que todos los registros en un archivo sean de la misma longitud fija. El uso de registros de longitud fija facilita a un programa calcular (como una función del tamaño y la clave del registro) la posición exacta de cualquier registro, en forma relativa al principio del archivo. Pronto veremos cómo esta capacidad facilita el acceso directo a registros específicos, incluso en archivos extensos.

En la figura 17.11 se ilustra la manera en que Java ve a un archivo de acceso aleatorio, compuesto de registros con longitud fija. (Cada registro en esta figura tiene 100 bytes de longitud.) Un archivo de acceso aleatorio es como un ferrocarril con muchos vagones; algunos están vacíos y otros están cargados.

Un programa puede insertar datos en un archivo de acceso aleatorio sin necesidad de destruir los demás datos en el archivo. Además, un programa puede actualizar o eliminar los datos almacenados previamente, sin necesidad de tener que reescribir todo el archivo completo. En las siguientes secciones explicaremos cómo crear un archivo de acceso aleatorio, cómo introducir datos, leerlos tanto en forma secuencial como aleatoria, actualizarlos y eliminarlos.

17.9 Creación de un archivo de acceso aleatorio

Los objetos `RandomAccessFile` tienen todas las capacidades de las clases `FileInputStream`, `FileOutputStream`, `DataInputStream` y `DataOutputStream`. Cuando un programa asocia a un objeto de la clase `RandomAccessFile` con un archivo, el programa lee o escribe datos, empezando en la posición, en el archivo, especificada por el *apuntador de posición de archivo*, y manipula a todos los datos como tipos primitivos. Al escribir un valor `int` se envían cuatro bytes al archivo. Al leer un valor `double`, se reciben ocho bytes del archivo. El tamaño de los tipos está garantizado, ya que Java tiene representaciones y tamaños fijos para todos los tipos primitivos, sin importar la plataforma computacional.

Los programas para procesar archivos de acceso aleatorio raras veces escriben un solo campo a un archivo. Generalmente escriben un objeto a la vez, como veremos en los siguientes ejemplos. Considere el siguiente problema:

> *Crear un programa para procesar transacciones que sea capaz de almacenar hasta 100 registros de longitud fija para una compañía que puede tener hasta 100 clientes. Cada registro debe consistir de un número de cuenta que se utilizará como la clave del registro, un apellido paterno, un primer nombre y un saldo. El programa debe ser capaz de actualizar una cuenta, insertar una nueva cuenta y eliminar una cuenta.*

En las siguientes secciones introduciremos las técnicas necesarias para crear este programa de procesamiento de créditos. En la figura 17.12 se muestra la clase **RegistroCuentasAccesoAleatorio** que es utilizada por los siguientes cuatro programas, tanto para leer registros de, y escribir registros en, un archivo. La clase **RegistroCuentasAccesoAleatorio** hereda la implementación de **RegistroCuentas** (figura 17.6), la cual incluye campos **private** (**cuenta**, **apellidoPaterno**, **primerNombre** y **saldo**) junto con métodos *establecer* y *obtener* para cada campo.

En la línea 9 se declara la constante **TAMANIO** para representar el tamaño (en bytes) de un registro. Un objeto **RandomAccessAccountRecord** contiene un valor **int** (4 bytes), dos cadenas que restringimos en 15 caracteres (30 bytes) cada una para este ejemplo, y un valor **double** (8 bytes) para un total de 72 bytes.

El método **leer** (líneas 25 a 31) lee un registro del objeto **RandomAccessFile** especificado como su argumento. Los métodos ***readInt*** (línea 27) y ***readDouble*** (línea 30) de **RandomAccessFile** leen la **cuenta** y el **balance**, respectivamente. El método **leer** llama al método utilitario **leerNombre** (líneas 34 a 44) dos veces para obtener el primer nombre y el apellido paterno. El método **leerNombre** lee 15 caracteres del objeto **RandomAccessFile** y devuelve un objeto **String**. Si la longitud de un nombre es menor de 15 caracteres, el programa rellena cada carácter adicional con un byte nulo (**'\0'**). Los componentes de Swing, como los objetos **JTextField**, no pueden mostrar caracteres nulos; en vez de ello, muestran esos caracteres como rectángulos). En la línea 43 se soluciona este problema, sustituyendo los bytes nulos con espacios.

El método **escribir** (líneas 47 a 53) escribe un registro en el objeto **RandomAccessFile** especificado como su argumento. Este método utiliza los siguientes métodos de **RandomAccessFile**: ***writeInt*** para escribir el valor entero de **cuenta**; el método ***writeChars*** (que es llamado desde el método utilitario **escribirNombre**) para escribir los arreglos de caracteres **primerNombre** y **apellidoPaterno**, y el método ***writeDouble*** para escribir el valor **double saldo**. (*Nota:* Para asegurar que todos los registros en el objeto **RandomAccessFile** tengan el mismo tamaño, escribimos exactamente 15 caracteres para el primer nombre y exactamente 15 caracteres para el apellido paterno.) El método **escribirNombre** (líneas 56 a 68) realiza las operaciones de escritura para el primer nombre y el apellido paterno. Observe que la clase se encuentra en el paquete **com.deitel.cpej5.cap17**.

```
1  // Fig. 17.12: RegistroCuentasAccesoAleatorio.java
2  // Subclase de RegistroCuentas para los programas que usan archivos de
   acceso aleatorio.
3  package com.deitel.cpej5.cap17;
4
5  import java.io.*;
6
7  public class RegistroCuentasAccesoAleatorio extends RegistroCuentas {
8
9     public static final int TAMANIO = 72;
10
11    // el constructor sin argumentos llama al otro constructor con los
         valores predeterminados
12    public RegistroCuentasAccesoAleatorio()
13    {
14       this( 0, "", "", 0.0 );
15    }
16
17    // inicializar un objeto RegistroCuentasAccesoAleatorio
18    public RegistroCuentasAccesoAleatorio( int cuenta, String primerNombre,
19       String apellidoPaterno, double saldo )
20    {
21       super( cuenta, primerNombre, apellidoPaterno, saldo );
22    }
23
24    // leer un registro del objeto RandomAccecssFile especificado
```

Figura 17.12 La clase **RegistroCuentasAccesoAleatorio** utilizada en los programas con archivos de acceso aleatorio. (Parte 1 de 2.)

http://librosysolucionarios.net


```
25    public void leer( RandomAccessFile archivo ) throws IOException
26    {
27       establecerCuenta( archivo.readInt() );
28       establecerPrimerNombre( leerNombre( archivo ) );
29       establecerApellidoPaterno( leerNombre( archivo ) );
30       establecerSaldo( archivo.readDouble() );
31    }
32
33    // asegurarse que el nombre sea de la longitud apropiada
34    private String leerNombre( RandomAccessFile archivo ) throws IOException
35    {
36       char nombre[] = new char[ 15 ], temp;
37
38       for ( int cuenta = 0; cuenta < nombre.length; cuenta++ ) {
39          temp = archivo.readChar();
40          nombre[ cuenta ] = temp;
41       }
42
43       return new String( nombre ).replace( '\0', ' ' );
44    }
45
46    // escribir un registro en el objeto RandomAccessFile especificado
47    public void escribir( RandomAccessFile archivo ) throws IOException
48    {
49       archivo.writeInt( obtenerCuenta() );
50       escribirNombre( archivo, obtenerPrimerNombre() );
51       escribirNombre( archivo, obtenerApellidoPaterno() );
52       archivo.writeDouble( obtenerSaldo() );
53    }
54
55    // escribir un nombre en el archivo; máximo 15 caracteres
56    private void escribirNombre( RandomAccessFile archivo, String nombre )
57       throws IOException
58    {
59       StringBuffer bufer = null;
60
61       if ( nombre != null )
62          bufer = new StringBuffer( nombre );
63       else
64          bufer = new StringBuffer( 15 );
65
66       bufer.setLength( 15 );
67       archivo.writeChars( bufer.toString() );
68    }
69
70 } // fin de la clase RegistroCuentasAccesoAleatorio
```

Figura 17.12 La clase **RegistroCuentasAccesoAleatorio** utilizada en los programas con archivos de acceso aleatorio. (Parte 2 de 2.)

En la figura 17.13 se ilustra el proceso de abrir un archivo de acceso aleatorio y escribir datos en el disco. Este programa escribe 100 objetos **RegistroCuentasAccesoAleatorio** en blanco. Cada objeto **RegistroCuentasAccesoAleatorio** contiene los valores de **0** en el número de cuenta, **null** en el apellido, **null** en el primer nombre y **0.0** en el saldo. El archivo se inicializa para crear la cantidad apropiada de espacio "vacío", en el cual se guardarán los datos de la cuenta. Esto nos permitirá determinar en los siguientes programas si un registro está vacío, o si contiene datos.

```
1  // Fig. 17.13: CrearArchivoAleatorio.java
2  // Crea un archivo de acceso aleatorio, escribiendo 100 registros vacíos en el disco.
3  import java.io.*;
4  import javax.swing.*;
5
6  import com.deitel.cpej5.cap17.RegistroCuentasAccesoAleatorio;
7
8  public class CrearArchivoAleatorio {
9
10    private static final int NUMERO_REGISTROS = 100;
11
12    // permitir al usuario seleccionar el archivo a abrir
13    private void crearArchivo()
14    {
15       // mostrar cuadro de diálogo para que el usuario pueda seleccionar el
             archivo
16       JFileChooser selectorArchivo = new JFileChooser();
17       selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );
18
19       int resultado = selectorArchivo.showSaveDialog( null );
20
21       // si el usuario hizo clic en el botón Cancelar del cuadro de
             diálogo, regresar
22       if ( resultado == JFileChooser.CANCEL_OPTION )
23          return;
24
25       // obtener el archivo seleccionado
26       File nombreArchivo = selectorArchivo.getSelectedFile();
27
28       // mostrar error si el nombre del archivo es inválido
29       if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) )
30          JOptionPane.showMessageDialog( null, "Nombre de archivo incorrecto",
31             "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
32
33       else {
34
35          // abrir el archivo
36          try {
37             RandomAccessFile archivo =
38                new RandomAccessFile( nombreArchivo, "rw" );
39
40             RegistroCuentasAccesoAleatorio registroEnBlanco =
41                new RegistroCuentasAccesoAleatorio();
42
43             // escribir 100 registros en blanco
44             for ( int cuenta = 0; cuenta < NUMERO_REGISTROS; cuenta++ )
45                registroEnBlanco.escribir( archivo );
46
47             archivo.close(); // cerrar el archivo
48
49             // mostrar mensaje indicando que el archivo se creó
50             JOptionPane.showMessageDialog( null, "Se creó el archivo " +
51                nombreArchivo, "Estado", JOptionPane.INFORMATION_MESSAGE );
52
53             System.exit( 0 );  // terminar el programa
54
55          } // fin del bloque try
```

Figura 17.13 Archivo de acceso aleatorio creado en forma secuencial. (Parte 1 de 2.)

http://librosysolucionarios.net


```
56
57          // procesar excepciones durante operaciones de apertura, escritura o
               cierre del archivo
58          catch ( IOException excepcionES ) {
59             JOptionPane.showMessageDialog( null, "Error al procesar el archivo",
60                "Error al procesar el archivo", JOptionPane.ERROR_MESSAGE );
61
62             System.exit( 1 );
63          }
64
65       } // fin de instrucción else
66
67    } // fin del método crearArchivo
68
69    public static void main( String args[] )
70    {
71       CrearArchivoAleatorio aplicacion = new CrearArchivoAleatorio();
72       aplicacion.crearArchivo();
73    }
74
75 } // fin de la clase CrearArchivoAleatorio
```

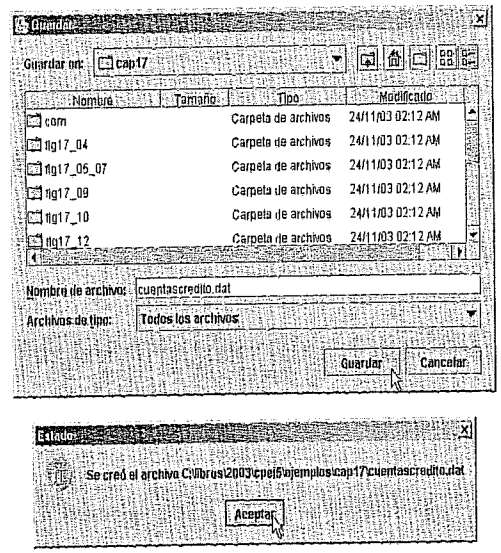


Figura 17.13 Archivo de acceso aleatorio creado en forma secuencial. (Parte 2 de 2.)

En las líneas 37 y 38 se trata de abrir un objeto **RandomAccessFile** para usarlo en este programa. El constructor de **RandomAccessFile** acepta dos argumentos: el nombre del archivo y el *modo de apertura del archivo*. El modo de apertura de archivo para un objeto **RandomAccessFile** puede ser **"r"** (para abrir el archivo en modo de lectura) o **"rw"** (para abrir el archivo en modo de lectura y escritura).

Si ocurre una excepción **IOException** durante el proceso de apertura del archivo, el programa muestra un cuadro de diálogo de mensaje y termina. Si el archivo se abre apropiadamente, el programa utiliza una estructura **for** (líneas 44 y 45) para invocar al método **escribir** de **RegistroCuentasAccesoAleatorio** 100 veces. Esta instrucción provoca que los campos del objeto **registroEnBlanco** se escriban en el archivo asociado con el objeto **archivo** de la clase **RandomAccessFile**.

17.10 Cómo escribir datos en forma aleatoria, en un archivo de acceso aleatorio

En la figura 17.14 se escriben datos en un archivo que se abre con el modo **"rw"** (para leer y escribir). En este ejemplo utilizamos el método ***seek*** de **RandomAccessFile** para determinar la posición exacta en el archivo, en donde se almacena un registro de información. El método **seek** establece el apuntador de posición de archivo en una posición específica en el archivo, relativa al principio del mismo, y el método **escribir** de **RegistroCuentasAccesoAleatorio** escribe los datos. Este programa supone que el usuario no va a introducir números de cuenta duplicados, y que va a escribir los datos apropiados en cada campo de texto.

```
1  // Fig. 17.14: EscribirArchivoAleatorio.java
2  // Este programa utiliza campos de texto para obtener información del usuario mediante el
3  // teclado, y escribe la información en un archivo de acceso aleatorio.
4  import java.awt.*;
5  import java.awt.event.*;
6  import java.io.*;
7  import javax.swing.*;
8
9  import com.deitel.cpej5.cap17.*;
10
11 public class EscribirArchivoAleatorio extends JFrame {
12    private RandomAccessFile salida;
13    private IUBanco interfazUsuario;
14    private JButton botonEntrar, botonAbrir;
15
16    private static final int NUMERO_REGISTROS = 100;
17
18    // configurar la GUI
19    public EscribirArchivoAleatorio()
20    {
21       super( "Escribir en archivo de acceso aleatorio" );
22
23       // crear instancia de la interfaz de usuario reutilizable IUBanco
24       interfazUsuario = new IUBanco( 4 );  // cuatro campos de texto
25       getContentPane().add( interfazUsuario,
26          BorderLayout.CENTER );
27
28       // obtener referencia al botón de tarea genérico hacerTarea1 en IUBanco
29       botonAbrir = interfazUsuario.obtenerBotonHacerTarea1();
30       botonAbrir.setText( "Abrir..." );
31
32       // registrar componente de escucha para llamar a abrirArchivo cuando se
            oprima el botón
33       botonAbrir.addActionListener(
34
35          // clase interna anónima para manejar evento de botonAbrir
36          new ActionListener() {
37
38             // permitir al usuario seleccionar el archivo a abrir
39             public void actionPerformed( ActionEvent evento )
40             {
41                abrirArchivo();
42             }
43
44          } // fin de la clase interna anónima
```

http://librosysolucionarios.net

Figura 17.14 Escribir datos en un archivo de acceso aleatorio. (Parte 1 de 5.)


```
45
46       ); // fin de la llamada a addActionListener
47
48       // registrar componente de escucha de ventana para el evento de cierre de
            ventana
49       addWindowListener(
50
51          // clase interna anónima para manejar evento windowClosing
52          new WindowAdapter() {
53
54             // agregar registro en la GUI, después cerrar el archivo
55             public void windowClosing( WindowEvent evento )
56             {
57                if ( salida != null )
58                   agregarRegistro();
59
60                cerrarArchivo();
61             }
62
63          } // fin de la clase interna anónima
64
65       ); // fin de la llamada a addWindowListener
66
67       // obtener referencia al botón de tarea genérico hacerTarea2 en IUBanco
68       botonEntrar = interfazUsuario.obtenerBotonHacerTarea2();
69       botonEntrar.setText( "Introducir" );
70       botonEntrar.setEnabled( false );
71
72       // registrar componente de escucha para llamar a agregarRegistro cuando se
            oprima el botón
73       botonEntrar.addActionListener(
74
75          // clase interna anónima para manejar evento de botonEntrar
76          new ActionListener() {
77
78             // agregar registro al archivo
79             public void actionPerformed( ActionEvent evento )
80             {
81                agregarRegistro();
82             }
83
84          } // fin de la clase interna anónima
85
86       ); // fin de la llamada a addActionListener
87
88       setSize( 300, 150 );
89       setVisible( true );
90    }
91
92    // permitir al usuario seleccionar el archivo a abrir
93    private void abrirArchivo()
94    {
95       // mostrar cuadro de diálogo para que el usuario pueda seleccionar el archivo
96       JFileChooser selectorArchivo = new JFileChooser();
97       selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );
98
99       int resultado = selectorArchivo.showOpenDialog( this );
```

Figura 17.14 Escribir datos en un archivo de acceso aleatorio. (Parte 2 de 5.)

```
100
101         // si el usuario hizo clic en el botón Cancelar del cuadro de diálogo, regresar
102         if ( resultado == JFileChooser.CANCEL_OPTION )
103            return;
104
105         // obtener el archivo seleccionado
106         File nombreArchivo = selectorArchivo.getSelectedFile();
107
108         // mostrar error si el nombre de archivo es incorrecto
109         if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) )
110            JOptionPane.showMessageDialog( this, "Nombre de archivo incorrecto",
111               "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
112
113         else {
114
115            // abrir el archivo
116            try {
117               salida = new RandomAccessFile( nombreArchivo, "rw" );
118               botonEntrar.setEnabled( true );
119               botonAbrir.setEnabled( false );
120            }
121
122            // procesar excepción que puede ocurrir al abrir el archivo
123            catch ( IOException excepcionES ) {
124               JOptionPane.showMessageDialog( this, "El archivo no existe",
125                  "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
126            }
127
128         } // fin de instrucción else
129
130      } // fin del método abrirArchivo
131
132      // cerrar el archivo y terminar la aplicación
133      private void cerrarArchivo()
134      {
135         // cerrar el archivo y salir
136         try {
137            if ( salida != null )
138               salida.close();
139
140            System.exit( 0 );
141         }
142
143         // procesar excepción que puede ocurrir al cerrar el archivo
144         catch( IOException excepcionES ) {
145            JOptionPane.showMessageDialog( this, "Error al cerrar el archivo",
146               "Error", JOptionPane.ERROR_MESSAGE );
147
148            System.exit( 1 );
149         }
150
151      } // fin del método cerrarArchivo
152
153      // agregar un registro al archivo
154      private void agregarRegistro()
```

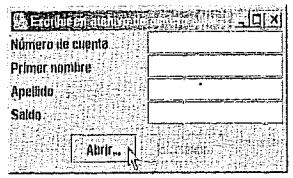
http://librosysolucionarios.net

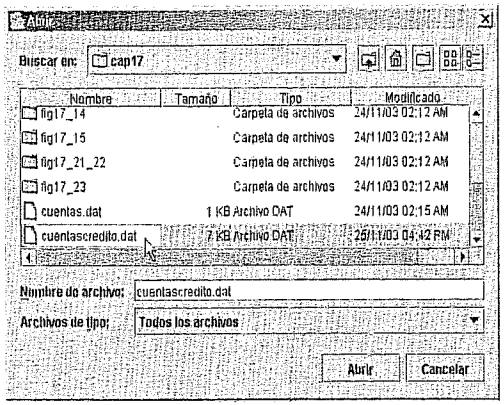
Figura 17.14 Escribir datos en un archivo de acceso aleatorio. (Parte 3 de 5.)


```
155      {
156         String campos[] = interfazUsuario.obtenerValoresCampos();
157
158         // asegurarse que el campo cuenta tenga un valor
159         if ( ! campos[ IUBanco.CUENTA ].equals( "" ) ) {
160
161            // escribir valores en el archivo
162            try {
163               int  numeroCuenta =
164                     Integer.parseInt( campos[ IUBanco.CUENTA ] );
165
166               if ( numeroCuenta > 0 && numeroCuenta <= NUMERO_REGISTROS ) {
167                     RegistroCuentasAccesoAleatorio registro =
168                           new RegistroCuentasAccesoAleatorio();
169
170                  registro.establecerCuenta( numeroCuenta );
171                  registro.establecerPrimerNombre( campos[ IUBanco.PRIMERNOMBRE ] );
172                  registro.establecerApellidoPaterno( campos[ IUBanco.APELLIDO ] );
173                  registro.establecerSaldo( Double.parseDouble(
174                     campos[ IUBanco.SALDO ] ) );
175
176                  salida.seek( ( numeroCuenta - 1 ) *
177                     RegistroCuentasAccesoAleatorio.TAMANIO );
178                  registro.escribir( salida );
179               }
180
181               else {
182                  JOptionPane.showMessageDialog( this,
183                     "El número de cuenta debe ser entre 0 y 100",
184                     "Número de cuenta incorrecto", JOptionPane.ERROR_MESSAGE );
185               }
186
187               interfazUsuario.borrarCampos(); // borrar el contenido de los campos de texto
188
189            } // fin del bloque try
190
191            // procesar formato incorrecto de número de cuenta o saldo
192            catch ( NumberFormatException excepcionFormato ) {
193               JOptionPane.showMessageDialog( this,
194                  "Número de cuenta o saldo incorrectos",
195                  "Formato de número incorrecto", JOptionPane.ERROR_MESSAGE );
196            }
197
198            // procesar excepciones que pueden ocurrir al escribir en el archivo
199            catch ( IOException excepcionES ) {
200               JOptionPane.showMessageDialog( this,
201                  "Error al escribir en el archivo", "Excepción de ES",
202                  JOptionPane.ERROR_MESSAGE );
203               cerrarArchivo();
204            }
205
206         } // fin de instrucción if
207
208      } // fin del método agregarRegistro
209
```

Figura 17.14 Escribir datos en un archivo de acceso aleatorio. (Parte 4 de 5.)

```
210     public static void main( String args[] )
211     {
212        new EscribirArchivoAleatorio();
213     }
214
215  } // fin de la clase EscribirArchivoAleatorio
```





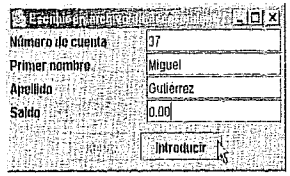


Figura 17.14 Escribir datos en un archivo de acceso aleatorio. (Parte 5 de 5.)

El usuario escribe valores para el número de cuenta, primer nombre, apellido paterno y saldo. Cuando el usuario hace clic en el botón **Introducir**, el programa llama al método **agregarRegistro** de **EscribirArchivoAleatorio** (líneas 154 a 208) para recuperar los datos de los campos de texto de **IUBanco**, almacena los datos en el objeto **registro** de **RegistroCuentasAccesoAleatorio** y llama al método **escribir** de **registro** para escribir los datos.

En las líneas 176 y 177 se hace una llamada al método ***seek*** de **RandomAccessFile** para posicionar el apuntador de posición de archivo para el objeto **salida** en la posición de byte calculada mediante **(numeroCuenta - 1) * RegistroCuentasAccesoAleatorio.TAMANIO**. Los números de cuenta en este programa deben estar entre 1 y 100. Restamos uno al número de cuenta para calcular la posición de byte del registro. Por lo tanto, para el primer registro el apuntador de posición de archivo se establece en el byte cero del archivo.

Cuando el usuario cierra la ventana, el programa trata de agregar el último registro al archivo (si hay uno en la GUI esperando a ser escrito en el archivo), cierra el archivo y termina su ejecución.

17.11 Cómo leer datos en forma secuencial, de un archivo de acceso aleatorio

En las secciones anteriores creamos un archivo de acceso aleatorio y escribimos datos en ese archivo. En esta sección desarrollaremos un programa (figura 17.15) que abre un objeto **RandomAccessFile** en modo de lectura mediante el modo de apertura de archivo **"r"**, lee a través del archivo en forma secuencial y muestra sólo los registros que contienen datos. Este programa produce un beneficio adicional. Vea si puede determinar cuál es; lo revelaremos al final de esta sección.



Buena práctica de programación 17.3

Abra un archivo con el modo de apertura de archivo "r" de lectura, si el contenido del archivo no debe modificarse. Esta práctica evita que se modifique accidentalmente el contenido del archivo. Es otro ejemplo del principio de menor privilegio.

```
1  // Fig. 17.15: LeerArchivoAleatorio.java
2  // Este programa lee un archivo de acceso aleatorio en forma secuencial y
3  // muestra el contenido de un registro a la vez, en campos de texto.
4  import java.awt.*;
5  import java.awt.event.*;
6  import java.io.*;
7  import java.text.DecimalFormat;
8  import javax.swing.*;
9
10 import com.deitel.cpej5.cap17.*;
11
12 public class LeerArchivoAleatorio extends JFrame {
13    private IUBanco interfazUsuario;
14    private RandomAccessFile entrada;
15    private JButton botonSiguiente, botonAbrir;
16
17    private static DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
18
19    // configurar la GUI
20    public LeerArchivoAleatorio()
21    {
22       super( "Leer archivo del cliente" );
23
24       // crear instancia de interfaz de usuario reutilizable
25       interfazUsuario = new IUBanco( 4 );  // cuatro campos de texto
26       getContentPane().add( interfazUsuario );
27
28       // configurar botón hacerTarea1 genérico de IUBanco
29       botonAbrir = interfazUsuario.obtenerBotonHacerTarea1();
30       botonAbrir.setText( "Abrir archivo para leer..." );
31
32       // registrar componente de escucha para llamar a archivoAbrir cuando se
            oprima el botón
33       botonAbrir.addActionListener(
34
35          // clase interna anónima para manejar evento de botonAbrir
36          new ActionListener() {
37
38             // permitir al usuario seleccionar el archivo a abrir
39             public void actionPerformed( ActionEvent evento )
40             {
```

Figura 17.15 Leer datos en forma secuencial de un archivo de acceso aleatorio. (Parte 1 de 5.)

http://librosysolucionarios.net

```
41                abrirArchivo();
42             }
43
44          } // fin de la clase interna anónima
45
46       ); // fin de la llamada a addActionListener
47
48       // configurar botón hacerTarea2 genérico de IUBanco
49       botonSiguiente = interfazUsuario.obtenerBotonHacerTarea2();
50       botonSiguiente.setText( "Siguiente" );
51       botonSiguiente.setEnabled( false );
52
53       // registrar componente de escucha para llamar a leerRegistro cuando se
            oprima el botón
54       botonSiguiente.addActionListener(
55
56          // clase interna anónima para manejar evento de botonSiguiente
57          new ActionListener() {
58
59             // leer un registro cuando el usuario haga clic en botonSiguiente
60             public void actionPerformed( ActionEvent evento )
61             {
62                leerRegistro();
63             }
64
65          } // fin de la clase interna anónima
66
67       ); // fin de la llamada a addActionListener
68
69       // registrar componente de escucha para evento de cierre de ventana
70       addWindowListener(
71
72          // clase interna anónima para manejar evento windowClosing
73          new WindowAdapter() {
74
75             // cerrar archivo y terminar la aplicación
76             public void windowClosing( WindowEvent evento )
77             {
78                cerrarArchivo();
79             }
80
81          } // fin de la clase interna anónima
82
83       ); // fin de la llamada a addWindowListener
84
85       setSize( 300, 150 );
86       setVisible( true );
87
88    } // fin del constructor
89
90    // permitir al usuario seleccionar el archivo a abrir
91    private void abrirArchivo()
92    {
93       // mostrar cuadro de diálogo para que el usuario pueda seleccionar el archivo
94       JFileChooser selectorArchivo = new JFileChooser();
95       selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );
```

Figura 17.15 Leer datos en forma secuencial de un archivo de acceso aleatorio. (Parte 2 de 5.)


```
96
97       int resultado = selectorArchivo.showOpenDialog( this );
98
99       // si el usuario hizo clic en el botón Cancelar del cuadro de diálogo, regresar
100      if ( resultado == JFileChooser.CANCEL_OPTION )
101         return;
102
103      // obtener el archivo seleccionado
104      File nombreArchivo = selectorArchivo.getSelectedFile();
105
106      // mostrar error si el nombre de archivo es incorrecto
107      if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) )
108         JOptionPane.showMessageDialog( this, "Nombre de archivo incorrecto",
109            "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
110
111      else {
112
113         // abrir el archivo
114         try {
115            entrada = new RandomAccessFile( nombreArchivo, "r" );
116            botonSiguiente.setEnabled( true );
117            botonAbrir.setEnabled( false );
118         }
119
120         // atrapar excepción que puede ocurrir al abrir el archivo
121         catch ( IOException ioException ) {
122            JOptionPane.showMessageDialog( this, "El archivo no existe",
123               "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
124         }
125
126      } // fin de instrucción else
127
128   } // fin del método abrirArchivo
129
130   // leer un registro
131   private void leerRegistro()
132   {
133      RegistroCuentasAccesoAleatorio registro
         = new RegistroCuentasAccesoAleatorio();
134
135      // leer un registro y mostrarlo
136      try {
137
138         do {
139            registro.leer( entrada );
140         } while ( registro.obtenerCuenta() == 0 );
141
142         String valores[] = { String.valueOf( registro.obtenerCuenta() ),
143            registro.obtenerPrimerNombre(), registro.obtenerApellidoPaterno(),
144            String.valueOf( registro.obtenerSaldo() ) };
145         interfazUsuario.establecerValoresCampos( valores );
146      }
147
148      // cerrar el archivo al llegar a su fin
149      catch ( EOFException excepcionEOF ) {
150         JOptionPane.showMessageDialog( this, "No hay más registros",
```

Figura 17.15 Leer datos en forma secuencial de un archivo de acceso aleatorio. (Parte 3 de 5.)

http://librosysolucionarios.net

```
151               "Se llegó al fin del archivo", JOptionPane.INFORMATION_MESSAGE );
152            cerrarArchivo();
153         }
154
155         // procesar excepciones que pueden ocurrir por algún problema con el archivo
156         catch ( IOException excepcionES ) {
157            JOptionPane.showMessageDialog( this, "Error al leer el archivo",
158               "Error", JOptionPane.ERROR_MESSAGE );
159
160            System.exit( 1 );
161         }
162
163      } // fin del método leerRegistro
164
165      // cerrar el archivo y terminar la aplicación
166      private void cerrarArchivo()
167      {
168         // cerrar el archivo y salir
169         try {
170            if ( entrada != null )
171               entrada.close();
172
173            System.exit( 0 );
174         }
175
176         // procesar excepción que puede ocurrir al cerrar el archivo
177         catch( IOException excepcionES ) {
178            JOptionPane.showMessageDialog( this, "Error al cerrar el archivo",
179               "Error", JOptionPane.ERROR_MESSAGE );
180
181            System.exit( 1 );
182         }
183
184      } // fin del método cerrarArchivo
185
186      public static void main( String args[] )
187      {
188         new LeerArchivoAleatorio();
189      }
190
191   } // fin de la clase LeerArchivoAleatorio
```

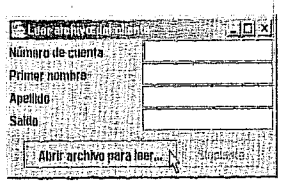


Figura 17.15 Leer datos en forma secuencial de un archivo de acceso aleatorio. (Parte 4 de 5.)

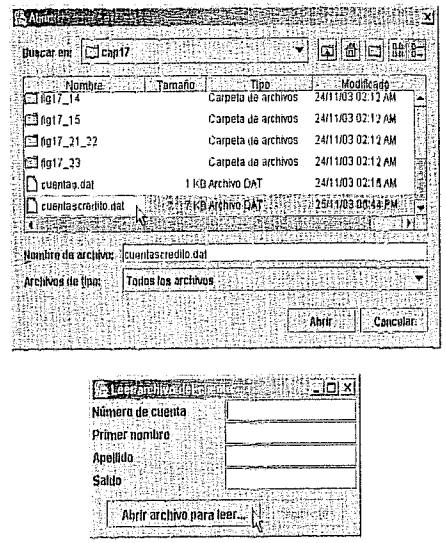


Figura 17.15 Leer datos en forma secuencial de un archivo de acceso aleatorio. (Parte 5 de 5.)

Cuando el usuario hace clic en el botón **Siguiente** para leer el siguiente registro en el archivo, el programa invoca al método `leerRegistro` de la clase `LeerArchivoAleatorio` (líneas 131 a 163). Este método invoca al método `leer` de `RegistroCuentasAccesoAleatorio` (línea 139) para leer los datos de un registro en el objeto `registro` tipo `RegistroCuentasAccesoAleatorio`. El método `leerRegistro` lee del archivo hasta encontrar un registro con un número de cuenta distinto de cero (cero es el número de cuenta para los registros vacíos), punto en el cual el ciclo termina y `leerRegistro` muestra los datos del registro en los campos de texto. Cuando el programa encuentra el marcador de fin de archivo mientras lee los datos o cuando hace clic en el botón con la **X** (en la esquina superior izquierda), se invoca el método `cerrarArchivo` para cerrar el archivo y terminar el programa.

¿Qué hay acerca de ese beneficio adicional que prometimos? Si analiza la GUI a medida que se ejecuta el programa, podrá observar que ¡los registros se muestran en orden ascendente (por número de cuenta)! Este ordenamiento es una simple consecuencia de la manera en que guardamos estos registros en el archivo, utilizando técnicas de acceso directo. En comparación con el método burbuja que hemos visto (capítulo 7), el ordenamiento con las técnicas de acceso directo es extremadamente rápido. La velocidad se logra haciendo que el archivo sea lo suficientemente grande como para guardar todos los posibles registros que podrían crearse, lo cual permite al programa insertar un registro entre otros registros sin tener que reorganizar el archivo. Desde luego que esta configuración significa que el archivo podría ocupar muy pocos lugares la mayor parte del tiempo, lo que sería un desperdicio de espacio. Esta situación es otro ejemplo de la concesión entre espacio/tiempo. Al utilizar grandes cantidades de espacio, podemos desarrollar un algoritmo de ordenamiento mucho más veloz.

17.12 Ejemplo práctico: Un programa para procesar transacciones

Ahora presentaremos un programa para procesar transacciones bastante substancial (figuras 17.21 y 17.22), utilizando un archivo de acceso aleatorio para lograr un procesamiento de acceso instantáneo. El programa mantiene la información sobre cuentas de banco. Actualiza las cuentas existentes, agrega nuevas cuentas y elimina las cuentas existentes. Supongamos que se ha ejecutado el programa de la figura 17.13 para crear un archivo, y que se ha ejecutado el programa de la figura 17.14 para insertar los datos iniciales. Hemos presentado las técnicas utilizadas en este ejemplo en los primeros ejemplos con `RandomAccessFile`.

http://librosysolucionarios.net

La GUI de este programa consiste en una ventana con una barra de menús, la cual contiene un menú **Archivo** y campos de texto que permiten al usuario realizar operaciones de inserción, actualización y eliminación de registros en el archivo. Se proporcionan botones para iniciar las acciones y para borrar los campos. El menú **Archivo** tiene cinco elementos de menú para seleccionar varias tareas, como se muestra en la figura 17.16.

Cuando el usuario selecciona **Actualizar registro**, **Eliminar registro** o **Nuevo registro** del menú **Archivo**, la GUI se actualiza para representar cada una de las vistas (figuras 17.17 a 17.20). Esto habilita ciertos campos de texto para editar, deshabilita a otros y cambia el texto del botón de acción. Si el usuario escribe un número en el campo de la cuenta y oprime la tecla *Intro*, en la línea 80 de la figura 17.21 se hace una llamada al método `mostrarRegistro` para introducir el registro en los campos de texto.

El método `realizarAccion` (figura 17.21, líneas 323 a 366) es responsable de tres acciones. Toma la información de los campos de texto y la envía al método apropiado de la clase `EditorArchivo` (figura 17.22), la cual encapsula las operaciones de procesamiento de archivos en este ejemplo. El método a llamar se determina mediante la comparación del argumento de cadena de `realizarAccion` con las cadenas `"Crear"`, `"Actualizar"` y `"Eliminar"`. Estas cadenas son de la etiqueta en el botón de acción en la GUI (el botón izquierdo en cada una de las capturas de pantalla de las figuras 17.17 a 17.20). El método `realizarAccion` también maneja cualquier excepción que pueda ser lanzada desde los métodos de `EditorArchivo`.

El método `mostrarRegistro` (figura 17.21, líneas 369 a 412) busca un registro del archivo y lo muestra en los campos de texto. Este método también toma una cadena como su argumento de entrada. Esta cadena

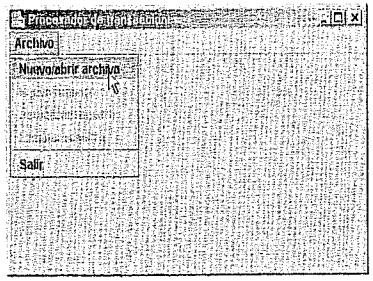


Figura 17.16 Ventana del **Procesador de transacciones**.

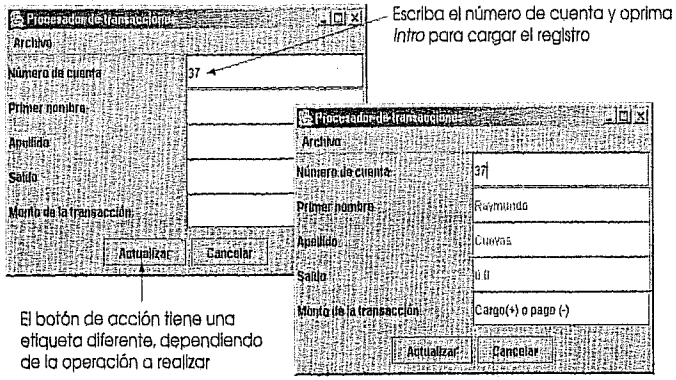


Figura 17.17 **Actualizar registro**: Cargar un registro para actualizarlo.

representa a una transacción. Cuando el método es llamado desde el componente de escucha de acciones del campo de la cuenta (línea 80), la cadena es `"0"`. Cuando el método es llamado desde el componente de escucha de acciones del campo de la transacción (línea 98), la cadena en el campo de la transacción se pasa como el argumento. (*Nota:* El campo de la transacción está habilitado sólo en el modo de actualización de registros.)

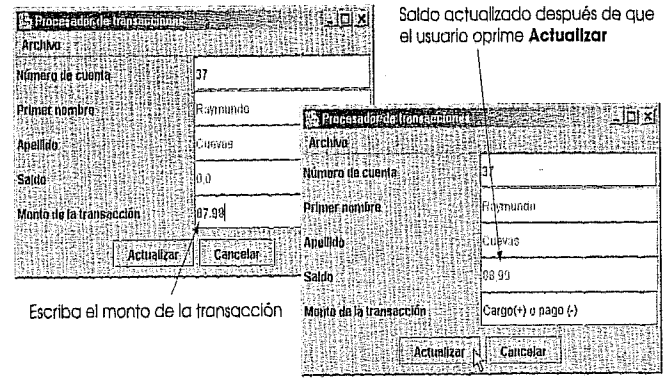


Figura 17.18 **Actualizar registro**: Introducción de una transacción.

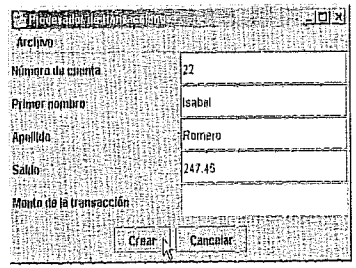


Figura 17.19 **Nuevo registro**: Agregar un registro al archivo.

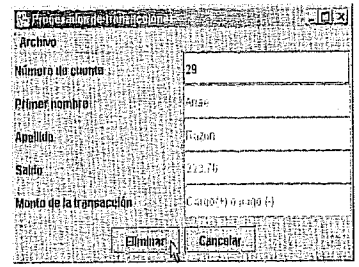


Figura 17.20 **Eliminar registro**: Eliminar un registro del archivo.

http://librosysolucionarios.net

```
1  // Fig. 17.21: ProcesadorTransacciones.java
2  // Un programa para procesar transacciones utilizando archivos de acceso aleatorio.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.text.DecimalFormat;
7  import javax.swing.*;
8
9  import com.deitel.cpej5.cap17.*;
10
11 public class ProcesadorTransacciones extends JFrame {
12
13    private IUBanco interfazUsuario;
14    private JMenuItem elementoNuevo, elementoActualizar, elementoEliminar,
      elementoAbrir, elementoSalir;
15    private JTextField campos[];
16    private JTextField campoCuenta, campoTransaccion;
17    private JButton botonAccion, botonCancelar;
18    private EditorArchivo archivoDatos;
19    private RegistroCuentasAccesoAleatorio registro;
20
21    public ProcesadorTransacciones()
22    {
23       super( "Procesador de transacciones" );
24
25       // establecer escritorio, barra de menús y menú Archivo
26       interfazUsuario = new IUBanco( 5 );
27       getContentPane().add( interfazUsuario );
28       interfazUsuario.setVisible( false );
29
30       // establecer el botón de acción
31       botonAccion = interfazUsuario.obtenerBotonHacerTarea1();
32       botonAccion.setText( "Guardar cambios" );
33       botonAccion.setEnabled( false );
34
35       // registrar componente de escucha para el botón de acción
36       botonAccion.addActionListener(
37
38          new ActionListener() { // clase interna anónima
39
40             public void actionPerformed( ActionEvent evento )
41             {
42                String accion = evento.getActionCommand();
43                realizarAccion( accion );
44
45             } // fin del método actionPerformed
46
47          } // fin de la clase interna anónima
48
49       ); // fin de la llamada a addActionListener
50
51       // establecer el botón para cancelar
52       botonCancelar = interfazUsuario.obtenerBotonHacerTarea2();
53       botonCancelar.setText( "Cancelar" );
54       botonCancelar.setEnabled( false );
55
```

Figura 17.21 Programa para procesar transacciones. (Parte 1 de 8.)


```
56       // registrar componente de escucha para el botón de cancelación
57       botonCancelar.addActionListener(
58
59          new ActionListener() { // clase interna anónima
60
61             // borrar los campos
62             public void actionPerformed( ActionEvent evento )
63             {
64                interfazUsuario.borrarCampos();
65             }
66
67          } // fin de la clase interna anónima
68
69       ); // fin de la llamada a addActionListener
70
71       // establecer el componente de escucha para el campo cuenta
72       campos = interfazUsuario.obtenerCampos();
73       campoCuenta = campos[ IUBanco.CUENTA ];
74       campoCuenta.addActionListener(
75
76          new ActionListener() { // clase interna anónima
77
78             public void actionPerformed( ActionEvent evento )
79             {
80                mostrarRegistro( "0" );
81             }
82
83          } // fin de la clase interna anónima
84
85       ); // fin de la llamada a addActionListener
86
87       // crear referencia al campo transaccion
88       campoTransaccion = campos[ IUBanco.TRANSACCION ];
89
90       // registrar componente de escucha para campo transaccion
91       campoTransaccion.addActionListener(
92
93          new ActionListener() { // clase interna anónima
94
95             // actualizar los campos de la GUI
96             public void actionPerformed( ActionEvent evento )
97             {
98                mostrarRegistro( campoTransaccion.getText() );
99             }
100
101         } // fin de la clase interna anónima
102
103      ); // fin de la llamada a addActionListener
104
105      JMenuBar barraMenus = new JMenuBar(); // establecer el menú
106      setJMenuBar( barraMenus );
107
108      JMenu menuArchivo = new JMenu( "Archivo" );
109      barraMenus.add( menuArchivo );
110
```

Figura 17.21 Programa para procesar transacciones. (Parte 2 de 8.)

http://librosysolucionarios.net

```
111        // establecer elemento de menú para agregar un registro
112        elementoNuevo = new JMenuItem( "Nuevo registro" );
113        elementoNuevo.setEnabled( false );
114
115        // registrar componente de escucha de elemento para nuevo registro
116        elementoNuevo.addActionListener(
117
118           new ActionListener() { // clase interna anónima
119
120              public void actionPerformed( ActionEvent evento )
121              {
122
123                 // configurar los campos de la GUI para su edición
124                 campos[ IUBanco.CUENTA ].setEnabled( true );
125                 campos[ IUBanco.PRIMERNOMBRE ].setEnabled( true );
126                 campos[ IUBanco.APELLIDO ].setEnabled( true );
127                 campos[ IUBanco.SALDO ].setEnabled( true );
128                 campos[ IUBanco.TRANSACCION ].setEnabled( false );
129
130                 botonAccion.setEnabled( true );
131                 botonAccion.setText( "Crear" );
132                 botonCancelar.setEnabled( true );
133
134                 interfazUsuario.borrarCampos(); // restablecer los campos de texto
135
136              } // fin del método actionPerformed
137
138           } // fin de la clase interna anónima
139
140        ); // fin de la llamada a addActionListener
141
142        // establecer elemento de menú para actualizar un registro
143        elementoActualizar = new JMenuItem( "Actualizar registro" );
144        elementoActualizar.setEnabled( false );
145
146        // registrar componente de escucha de elemento para actualizar
147        elementoActualizar.addActionListener(
148
149           new ActionListener() { // clase interna anónima
150
151              public void actionPerformed( ActionEvent evento )
152              {
153                 // configurar los campos de la GUI para su edición
154                 campos[ IUBanco.CUENTA ].setEnabled( true );
155                 campos[ IUBanco.PRIMERNOMBRE ].setEnabled( false );
156                 campos[ IUBanco.APELLIDO ].setEnabled( false );
157                 campos[ IUBanco.SALDO ].setEnabled( false );
158                 campos[ IUBanco.TRANSACCION ].setEnabled( true );
159
160                 botonAccion.setEnabled( true );
161                 botonAccion.setText( "Actualizar" );
162                 botonCancelar.setEnabled( true );
163
164                 interfazUsuario.borrarCampos(); // restablecer los campos de texto
165
```

http://librosysolucionarios.net

Figura 17.21 Programa para procesar transacciones. (Parte 3 de 8.)


```
166              } // fin del método actionPerformed
167
168           } // fin de la clase interna anónima
169
170        ); // fin de la llamada a addActionListener
171
172        // establecer elemento de menú para eliminar un registro
173        elementoEliminar = new JMenuItem( "Eliminar registro" );
174        elementoEliminar.setEnabled( false );
175
176        // registrar componente de escucha para elemento eliminar
177        elementoEliminar.addActionListener(
178
179           new ActionListener() { // clase interna anónima
180
181              public void actionPerformed( ActionEvent evento )
182              {
183                 // configurar los campos de la GUI para su edición
184                 campos[ IUBanco.CUENTA ].setEnabled( true );
185                 campos[ IUBanco.PRIMERNOMBRE ].setEnabled( false );
186                 campos[ IUBanco.APELLIDO ].setEnabled( false );
187                 campos[ IUBanco.SALDO ].setEnabled( false );
188                 campos[ IUBanco.TRANSACCION ].setEnabled( false );
189
190                 botonAccion.setEnabled( true );
191                 botonAccion.setText( "Eliminar" );
192                 botonCancelar.setEnabled( true );
193
194                 interfazUsuario.borrarCampos(); // restablecer los campos de texto
195
196              } // fin del método actionPerformed
197
198           } // fin de la clase interna anónima
199
200        ); // fin de la llamada a addActionListener
201
202        // establecer elemento de menú para abrir archivo
203        elementoAbrir = new JMenuItem( "Nuevo/abrir archivo" );
204
205        // registrar componente de escucha para elemento abrir
206        elementoAbrir.addActionListener(
207
208           new ActionListener() { // clase interna anónima
209
210              public void actionPerformed( ActionEvent evento )
211              {
212                 // tratar de abrir el archivo
213                 if ( !abrirArchivo() )
214                    return;
215
216                 // establecer los elementos del menú
217                 elementoNuevo.setEnabled( true );
218                 elementoActualizar.setEnabled( true );
219                 elementoEliminar.setEnabled( true );
220                 elementoAbrir.setEnabled( false );
```

Figura 17.21 Programa para procesar transacciones. (Parte 4 de 8.)

```
            // establecer la interfaz
            interfazUsuario.setVisible( true );
            campos[ IUBanco.CUENTA ].setEnabled( false );
            campos[ IUBanco.PRIMERNOMBRE ].setEnabled( false );
            campos[ IUBanco.APELLIDO ].setEnabled( false );
            campos[ IUBanco.SALDO ].setEnabled( false );
            campos[ IUBanco.TRANSACCION ].setEnabled( false );

         } // fin del método actionPerformed

      } // fin de la clase interna anónima

   ); // fin de la llamada a addActionListener

   // establecer elemento de menú para salir del programa
   elementoSalir = new JMenuItem( "Salir" );

   // registrar componente de escucha para elemento salir
   elementoSalir.addActionListener(

      new ActionListener() { // clase interna anónima

         public void actionPerformed( ActionEvent evento )
         {
            try {
               archivoDatos.cerrarArchivo(); // cerrar el archivo
            }

            catch ( IOException excepcionES ) {
               JOptionPane.showMessageDialog(
                  ProcesadorTransacciones.this, "Error al cerrar el archivo",
                  "Error de ES", JOptionPane.ERROR_MESSAGE );
            }

            finally {
               System.exit( 0 ); // salir del programa
            }

         } // fin del método actionPerformed

      } // fin de la clase interna anónima

   ); // fin de la llamada a addActionListener

   // adjuntar elementos de menú al menú Archivo
   menuArchivo.add( elementoAbrir );
   menuArchivo.add( elementoNuevo );
   menuArchivo.add( elementoActualizar );
   menuArchivo.add( elementoEliminar );
   menuArchivo.addSeparator();
   menuArchivo.add( elementoSalir );

   setSize( 400, 250 );
   setVisible( true );
```

Figura 17.21 Programa para procesar transacciones. (Parte 5 de 8.)


```
   } // fin del constructor

public static void main( String args[] )
{
   new ProcesadorTransacciones();
}

// obtener el nombre del archivo y abrirlo
private boolean abrirArchivo()
{
   // mostrar cuadro de diálogo para que el usuario pueda seleccionar el archivo
   JFileChooser selectorArchivo = new JFileChooser();
   selectorArchivo.setFileSelectionMode( JFileChooser.FILES_ONLY );

   int resultado = selectorArchivo.showOpenDialog( this );

   // si el usuario hizo clic en el botón Cancelar del cuadro de diálogo, regresar
   if ( resultado == JFileChooser.CANCEL_OPTION )
      return false;

   // obtener el archivo seleccionado
   File nombreArchivo = selectorArchivo.getSelectedFile();

   // mostrar error si el nombre de archivo es incorrecto
   if ( nombreArchivo == null || nombreArchivo.getName().equals( "" ) ) {
      JOptionPane.showMessageDialog( this, "Nombre de archivo incorrecto",
         "Nombre de archivo incorrecto", JOptionPane.ERROR_MESSAGE );
      return false;
   }

   try {
      // llamar al método ayudante para abrir el archivo
      archivoDatos = new EditorArchivo( nombreArchivo );
   }

   catch( IOException excepcionES ) {
      JOptionPane.showMessageDialog( this, "Error al abrir el archivo",
         "Error de ES", JOptionPane.ERROR_MESSAGE );
      return false;
   }

   return true;

} // fin del método abrirArchivo

// crear, actualizar o eliminar el registro
private void realizarAccion( String accion )
{
   try {

      // obtener los valores de los campos de texto
      String[] valores = interfazUsuario.obtenerValoresCampos();

      int numeroCuenta = Integer.parseInt( valores[ IUBanco.CUENTA ] );
```

Figura 17.21 Programa para procesar transacciones. (Parte 6 de 8.)

http://librosysolucionarios.net

```
331          String primerNombre = valores[ IUBanco.PRIMERNOMBRE ];
332          String apellidoPaterno = valores[ IUBanco.APELLIDO ];
333          double saldo = Double.parseDouble( valores[ IUBanco.SALDO ] );
334
335          if ( accion.equals( "Crear" ) )
336             archivoDatos.nuevoRegistro( numeroCuenta, // crear un nuevo registro
337                primerNombre, apellidoPaterno, saldo );
338
339          else if ( accion.equals( "Actualizar" ) )
340             archivoDatos.actualizarRegistro( numeroCuenta, // actualizar registro
341                primerNombre, apellidoPaterno, saldo );
342
343          else if ( accion.equals( "Eliminar" ) )
344             archivoDatos.eliminarRegistro( numeroCuenta ); // eliminar registro
345
346          else
347             JOptionPane.showMessageDialog( this, "Acción incorrecta",
348                "Error al ejecutar la acción", JOptionPane.ERROR_MESSAGE );
349
350       } // fin del bloque try
351
352       catch( NumberFormatException formato ) {
353          JOptionPane.showMessageDialog( this, "Entrada incorrecta",
354             "Error en formato de número", JOptionPane.ERROR_MESSAGE );
355       }
356
357       catch( IllegalArgumentException cuentaIncorrecta ) {
358          JOptionPane.showMessageDialog( this, cuentaIncorrecta.getMessage(),
359             "Número de cuenta incorrecto", JOptionPane.ERROR_MESSAGE );
360       }
361       catch( IOException excepcionES ) {
362          JOptionPane.showMessageDialog( this, "Error al escribir en el archivo",
363             "Error de ES", JOptionPane.ERROR_MESSAGE );
364       }
365
366    } // fin del método realizarAccion
367
368    //  escribir un registro en los campos de texto y actualizar el saldo
369    private void mostrarRegistro( String transaccion )
370    {
371       try {
372          // obtener el número de cuenta
373          int numeroCuenta = Integer.parseInt(
374             interfazUsuario.obtenerValoresCampos()[ IUBanco.CUENTA ] );
375
376          // obtener el registro asociado
377          RegistroCuentasAccesoAleatorio registro =
378             archivoDatos.obtenerRegistro( numeroCuenta );
379
380          if ( registro.obtenerCuenta() == 0 )
381             JOptionPane.showMessageDialog( this, "El registro no existe",
382                "Número de cuenta incorrecto",  JOptionPane.ERROR_MESSAGE );
383
384          // obtener la transaccion
385          double cambiar = Double.parseDouble( transaccion );
```

Figura 17.21 Programa para procesar transacciones. (Parte 7 de 8.)

http://librosysolucionarios.net


```
386
387          // crear un arreglo de cadena para enviarlo a los campos de texto
388          String[] valores = { String.valueOf( registro.obtenerCuenta() ),
389             registro.obtenerPrimerNombre(), registro.obtenerApellidoPaterno(),
390             String.valueOf( registro.obtenerSaldo() + cambiar ),
391             "Cargo(+) o pago (-)" };
392
393          interfazUsuario.establecerValoresCampos( valores );
394
395       } // fin del bloque try
396
397       catch( NumberFormatException formato ) {
398          JOptionPane.showMessageDialog( this, "Entrada incorrecta",
399             "Error en formato de número", JOptionPane.ERROR_MESSAGE );
400       }
401
402       catch ( IllegalArgumentException cuentaIncorrecta ) {
403          JOptionPane.showMessageDialog( this, cuentaIncorrecta.getMessage(),
404             "Número de cuenta incorrecto", JOptionPane.ERROR_MESSAGE );
405       }
406
407       catch( IOException excepcionES ) {
408          JOptionPane.showMessageDialog( this, "Error al leer el archivo",
409             "Error de ES", JOptionPane.ERROR_MESSAGE );
410       }
411
412    } // fin del método mostrarRegistro
413
414 } // fin de la clase ProcesadorTransacciones
```

Figura 17.21 Programa para procesar transacciones. (Parte 8 de 8.)

La clase **EditorArchivo** (figura 17.22) declara métodos para manipular los registros en un archivo de acceso aleatorio. El método **obtenerRegistro** (líneas 26 a 41) lee el registro con el número de cuenta dado y almacena su información en un objeto **RegistroCuentasAccesoAleatorio**. El método **actualizarRegistro** (líneas 44 a 60) modifica el registro con el número de cuenta dado, utilizando los valores introducidos para **primerNombre**, **apellidoPaterno** y **saldo**. El método **nuevoRegistro** (líneas 63 a 80) agrega un nuevo registro al archivo, utilizando los valores proporcionados para el número de cuenta, primer nombre, apellido paterno y saldo. El método **eliminarRegistro** (líneas 83 a 98) elimina del archivo el registro con el número de cuenta dado.

```
1  // Fig. 17.22: EditorArchivo.java
2  // Esta clase declara métodos para manipular los registros de una
3  // cuenta de banco en un archivo de acceso aleatorio.
4  import java.io.*;
5
6  import com.deitel.cpej5.cap17.RegistroCuentasAccesoAleatorio;
7
8  public class EditorArchivo {
9
10    RandomAccessFile archivo; // referencia al archivo
11
```

Figura 17.22 La clase **EditorArchivo** que encapsula las herramientas para procesar archivos requeridas en la figura 17.21. (Parte 1 de 3.)

```
12    // abrir el archivo
13    public EditorArchivo( File nombreArchivo ) throws IOException
14    {
15       archivo = new RandomAccessFile( nombreArchivo, "rw" );
16    }
17
18    // cerrar el archivo
19    public void cerrarArchivo() throws IOException
20    {
21       if ( archivo != null )
22          archivo.close();
23    }
24
25    // obtener un registro del archivo
26    public RegistroCuentasAccesoAleatorio obtenerRegistro( int numeroCuenta )
27       throws IllegalArgumentException, NumberFormatException, IOException
28    {
29       RegistroCuentasAccesoAleatorio registro = new
         RegistroCuentasAccesoAleatorio();
30
31       if ( numeroCuenta < 1 || numeroCuenta > 100 )
32          throw new IllegalArgumentException( "Fuera de rango" );
33
34       // buscar registro apropiado en el archivo
35       archivo.seek( ( numeroCuenta - 1 ) * RegistroCuentasAccesoAleatorio.TAMANIO );
36
37       registro.leer( archivo );
38
39       return registro;
40
41    } // fin del método obtenerRegistro
42
43    // actualizar registro en el archivo
44    public void actualizarRegistro( int numeroCuenta, String primerNombre,
45       String apellidoPaterno, double saldo )
46       throws IllegalArgumentException, IOException
47    {
48       RegistroCuentasAccesoAleatorio registro = obtenerRegistro( numeroCuenta );
49       if ( numeroCuenta == 0 )
50          throw new IllegalArgumentException( "La cuenta no existe" );
51
52       // buscar registro apropiado en el archivo
53       archivo.seek( ( numeroCuenta - 1 ) * RegistroCuentasAccesoAleatorio.TAMANIO );
54
55       registro = new RegistroCuentasAccesoAleatorio( numeroCuenta,
56          primerNombre, apellidoPaterno, saldo );
57
58       registro.escribir( archivo ); // escribir registro actualizado en el archivo
59
60    } // fin del método actualizarRegistro
61
62    // agregar registro al archivo
63    public void nuevoRegistro( int numeroCuenta, String primerNombre,
64       String apellidoPaterno, double saldo )
65       throws IllegalArgumentException, IOException
```

Figura 17.22 La clase **EditorArchivo** que encapsula las herramientas para procesar archivos requeridas en la figura 17.21. (Parte 2 de 3.)


```
66    {
67       RegistroCuentasAccesoAleatorio registro = obtenerRegistro( numeroCuenta );
68
69       if ( registro.obtenerCuenta() != 0 )
70          throw new IllegalArgumentException( "La cuenta ya existe" );
71
72        // buscar registro apropiado en el archivo
73       archivo.seek( ( numeroCuenta - 1 ) * RegistroCuentasAccesoAleatorio.TAMANIO );
74
75       registro = new RegistroCuentasAccesoAleatorio( numeroCuenta,
76          primerNombre, apellidoPaterno, saldo );
77
78       registro.escribir( archivo ); // escribir registro en el archivo
79
80    } // fin del método nuevoRegistro
81
82    // eliminar registro del archivo
83    public void eliminarRegistro( int numeroCuenta )
84       throws IllegalArgumentException, IOException
85    {
86       RegistroCuentasAccesoAleatorio registro = obtenerRegistro( numeroCuenta );
87
88       if ( registro.obtenerCuenta() == 0 )
89          throw new IllegalArgumentException( "La cuenta no existe" );
90
91       // buscar registro apropiado en el archivo
92       archivo.seek( ( numeroCuenta - 1 ) * RegistroCuentasAccesoAleatorio.TAMANIO );
93
94       // crear un registro en blanco para escribir en el archivo
95       registro = new RegistroCuentasAccesoAleatorio();
96       registro.escribir( archivo );
97
98    } // fin del método eliminarRegistro
99
100 } // fin de la clase EditorArchivo
```

Figura 17.22 La clase **EditorArchivo** que encapsula las herramientas para procesar archivos requeridas en la figura 17.21. (Parte 3 de 3.)

17.13 Nuevas APIs de E/S para la plataforma Java

La versión más reciente de Java (versión 1.4) introduce muchas nuevas características. Tal vez las adiciones más importantes sean las nuevas APIs de E/S incluidas en el paquete ***java.nio***, que mejoran la eficiencia de las operaciones de E/S. Los principales conceptos de las nuevas APIs de E/S son: búferes, canales, expresiones regulares, bloqueos de archivos, conjuntos de caracteres y E/S sin bloqueo. En la sección 11.8 introdujimos las expresiones regulares. Aquí hablaremos sobre los búferes, canales, bloqueos de archivos y conjuntos de caracteres. En el capítulo 18, Redes, hablaremos sobre la E/S sin bloqueo.

Búferes

Las operaciones de entrada y salida son considerablemente más lentas que las demás operaciones. Los *búferes* agilizan las operaciones de entrada y salida mediante la combinación de muchas operaciones de E/S lentas y pequeñas en una sola operación de E/S extensa. Al leer datos, se lee una gran parte de ellos y se coloca en un búfer. El programa utiliza esos datos en partes más pequeñas, sin necesidad de llamar a varias operaciones de entrada. Al escribir datos, se escriben varias piezas de los mismos en un búfer. Ese búfer puede escribirse mediante una sola operación. Los búferes sustituyen muchas operaciones de E/S lentas con muchas operaciones

http://librosysolucionarios.net

rápidas en el búfer y una sola operación de E/S lenta. Un búfer es similar a un arreglo, en cuanto a que contiene un número fijo de elementos de un cierto tipo primitivo. De hecho, muchos búferes almacenan sus datos en un *arreglo de respaldo*.

Cada búfer tiene cuatro propiedades: *capacidad*, *límite*, *posición* y *marca*. La *capacidad* del búfer indica cuántos datos puede almacenar. Este valor se establece al crear el búfer y no puede cambiarse. Debe ser un entero no negativo.

El *límite* de un búfer es su parte final. El límite permite cambiar el tamaño de un búfer sin tener que crear uno nuevo. No se pueden leer ni escribir elementos más allá de ese valor. Esta propiedad es útil para los búferes que se reutilizan. Por ejemplo, un búfer con capacidad de 100 se llena de datos. El programa después reutiliza el búfer, pero solamente escribe 10 nuevos elementos en él. El límite puede establecerse en 10, de manera que no se tenga acceso a los datos del uso anterior. El límite siempre debe ser un valor entero no negativo, pero puede ser mayor a la capacidad del búfer.

La *posición* de un búfer indica el siguiente elemento que se leerá o escribirá. Este valor cambia cada vez que se lee o se escribe un elemento en un búfer. La posición debe ser un entero no negativo y siempre debe ser menor o igual al límite del búfer.

La *marca* es un lugar que se recuerda en el búfer. El valor de la marca es indefinido cuando se crea el búfer, pero puede establecerse después. La marca debe ser siempre un entero no negativo y debe ser menor o igual a la posición. La marca permite al programador regresar a un punto anterior en el búfer. La marca se utiliza para recordar una posición en el búfer, a la que el programa podría regresar después.

Existen ocho clases de búferes. El búfer más simple es `ByteBuffer`. Los objetos `ByteBuffer` son secuencias de bytes. Un ejemplo de esto es un *arreglo de bytes envuelto*, que es un `ByteBuffer` implementado como arreglo de bytes. Con un *búfer directo*, la Máquina virtual de Java trata de usar las operaciones de E/S nativas del sistema operativo directamente en el búfer, lo cual hace que estas operaciones sean más rápidas. Los búferes directos consumen más recursos, por lo que deben utilizarse solamente cuando el búfer es grande, cuando va a tener una larga duración o cuando se va a reutilizar con frecuencia.

También existen búferes para cada uno de los tipos primitivos, excepto `boolean`. Están las clases `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` y `ShortBuffer`. El último búfer es la clase `MappedByteBuffer`, que es un tipo de `ByteBuffer` directo, el cual se asigna directamente a una región de un archivo.

Los datos se escriben en un búfer mediante una operación *colocar (put)*, y se leen de un búfer mediante una operación *obtener (get)*. Estas operaciones pueden ser *relativas* o *absolutas*. Una operación *obtener* o *colocar* empieza con el elemento que se encuentra en la posición actual del búfer. Si la operación se completa correctamente, la posición se actualiza en el elemento que se encuentra después del último elemento que se leyó o se escribió. Una operación *obtener* o *colocar* toma el índice del elemento inicial para esa operación como un argumento adicional, y no actualiza la posición del búfer.

Los búferes tienen cuatro operaciones más que alteran el estado del búfer. Ninguna de estas operaciones modifica los datos en el búfer; sólo afectan a las cuatro propiedades del búfer. Una operación *borrar* establece la posición del búfer en 0 y el límite en la capacidad. Esta operación se utiliza a menudo para que el búfer esté listo para leer o escribir más datos. Una operación *voltear (flip)* establece el límite en la posición actual y establece la posición en 0. Una vez que los datos se han escrito en un búfer con una operación "colocar" relativa, la posición apuntará al elemento que sigue después del último elemento escrito. Una operación "voltear" establece el límite en la posición (el fin de los datos) y la posición en 0 (el principio de los datos), para que el búfer esté listo para leer. Una operación *regresar (rewind)* establece la posición en 0, sin alterar el límite. Esta operación se utiliza a menudo para preparar un búfer, de manera que los datos puedan volver a leerse. Una operación *restablecer (reset)* establece la posición en la marca. Esta operación se utiliza para regresar a un punto anterior en el búfer.

Canales

La segunda parte de la nueva API de E/S introduce a los *canales*. Un canal es una conexión a un dispositivo de E/S, el cual está diseñado para interactuar eficientemente con los búferes. La interfaz `Channel` declara solamente dos métodos: `close` e `isOpen`. Un canal se abre al crearse, y no puede volver a abrirse una vez cerrado.

Las dos subinterfaces más importantes de `Channel` son `ReadableByteChannel` y `WritableByteChannel`. La interfaz `ReadableByteChannel` declara el método `read`; la interfaz `WritableByteChannel` declara el método `write`. La interfaz `ByteChannel` es una interfaz de conveniencia que extiende tanto a `ReadableByteChannel` como a `WritableByteChannel` y, por lo tanto, permite tanto la lectura como la escritura. El método `read` toma un argumento `ByteBuffer`, el cual se rellena con bytes provenientes del canal. El método `write` toma un objeto `ByteBuffer`, cuyo contenido se escribe en el canal.

La interfaz `ScatteringByteChannel` implementa a `ReadableByteChannel` y permite *lecturas esparcidas*. Una lectura esparcida es cuando se llenan varios búferes con una sola operación de lectura. Por ejemplo, suponga que un programa necesita leer dos valores enteros, tres caracteres y un valor `double` de un canal. En vez de leer dos valores `int` en un `IntBuffer`, después leer tres valores `char` en un `CharBuffer` y por último leer un valor `double` en un `DoubleBuffer`, el programa puede realizar las tres operaciones de lectura de una sola vez, mediante una lectura esparcida. La interfaz `GatheringByteChannel` implementa a `WritableByteChannel` y permite *escrituras esparcidas*. Una escritura esparcida toma datos de varios búferes y los escribe en el canal, en una sola operación. Las lecturas y escrituras esparcidas agilizan aún más las operaciones de entrada y de salida.

La clase `FileChannel` representa un canal conectado a un archivo. Esta clase implementa a `ByteChannel`, `ScatteringByteChannel` y `GatheringByteChannel`. Permite operaciones de lectura, de escritura, lecturas y escrituras esparcidas. Un objeto `FileChannel` se crea mediante una llamada al método `getChannel` en un objeto `FileInputStream`, un objeto `FileOutputStream` o un objeto `RandomAccessFile`. Los objetos `SocketChannel` y `ServerSocketChannel` permiten la comunicación de red basada en sockets. Describiremos a estas clases en el capítulo 18.

Bloqueos de archivos

Un *bloqueo de archivo* restringe el acceso a una porción de un archivo. Cada bloqueo de archivo tiene un canal de archivo, una posición dentro del archivo y un tamaño asociados. El canal del archivo es un conducto a un archivo físico en el disco. Éste es el archivo que va a bloquearse. La posición es el byte inicial que pertenece al bloqueo, y el tamaño es el número de bytes que pertenecen al bloqueo. Estas dos variables definen la porción del archivo que pertenece a ese bloqueo. Se dice que dos bloqueos se *traslapan* si hay un byte que pertenece a ambos bloqueos.

Un bloqueo de archivo puede ser *excluyente* o *compartido*. Un bloqueo excluyente no permite que se creen bloqueos traslapados en su porción del archivo. Los bloqueos excluyentes se crean generalmente por procesos que escriben en un archivo. Los bloqueos compartidos pueden traslapar porciones de un archivo, pero no permiten que un bloqueo excluyente se traslape en esa porción. Los bloqueos compartidos permiten que varios procesos lean el mismo archivo.

Hay dos métodos que pueden ser llamados por un objeto `FileChannel` para adquirir un bloqueo de archivo: `lock` y `tryLock`. El método `lock` es una operación *bloqueadora*. Esto significa que el método regresará solamente cuando se haya adquirido el bloqueo. El método `tryLock` es una operación *no bloqueadora*. Este método regresa inmediatamente cuando es llamado, pero no garantiza que se haya adquirido el bloqueo. Si no puede adquirirse el bloqueo, el método devuelve `null`. Hablaremos sobre otras operaciones no bloqueadoras con más detalle en el capítulo 18.

Observación de ingeniería de software 17.2



Los sistemas operativos manejan los bloqueos de archivo por proceso, y no por subproceso. Esto significa que los bloqueos de archivos no pueden utilizarse para coordinar el acceso a un archivo entre varios subprocesos de Java. Debe utilizarse un mecanismo distinto.

Conjuntos de caracteres

Un carácter es un miembro de un conjunto de símbolos que se combinan para expresar información. Un *conjunto codificado de caracteres* es una asignación de valores numéricos a un conjunto de caracteres. Unicode es un ejemplo de ello, y es el conjunto utilizado por Java. Esto significa que el carácter `'A'` se asigna al número 65, el carácter `'r'` se asigna al número 114 y el carácter `'ñ'` se asigna al número 257. Un *esquema de codificación de caracteres* es la asignación de los miembros de un conjunto codificado de caracteres a una se-

http://librosysolucionarios.net

cuencia de bytes. Un *conjunto de caracteres* es la combinación entre un conjunto codificado de caracteres y un esquema de codificación de caracteres. Un conjunto de caracteres asigna una secuencia de bytes a una secuencia de caracteres.

Unicode no es la única manera de asignar caracteres. Durante muchos años, el estándar en los Estados Unidos fue el conjunto codificado de caracteres ASCII. Este conjunto codificado de caracteres contiene solamente los caracteres que se utilizan en el lenguaje inglés, y ciertos caracteres especiales (por ejemplo, los caracteres de nueva línea y tabulador). Algunos archivos de caracteres están guardados todavía con un conjunto de caracteres ASCII, otros con Unicode y unos cuantos más con un conjunto de caracteres distinto. A menudo es difícil realizar convenciones entre dos o más conjuntos de caracteres.

La nueva API de E/S contiene el paquete `java.nio.charset` para encargarse de este problema. La mayor parte de la funcionalidad se encuentra en tres clases. Un objeto de la clase `Charset` representa a un conjunto de caracteres específico. El método `decode` convierte los bytes en el conjunto de caracteres (se recibe de entrada como un objeto `ByteBuffer`) en un objeto `CharBuffer` que contiene caracteres Unicode. Dos métodos `encode` convierten a un objeto `CharBuffer` o a una cadena de caracteres Unicode en bytes, en el objeto `Charset` (se envía de salida como un objeto `ByteBuffer`).

Las otras dos clases en este paquete son `CharsetDecoder` y `CharsetEncoder`. La clase `CharsetDecoder` representa a un objeto decodificador para transformar los bytes de un conjunto de caracteres específico en caracteres Unicode. Declara el método `decode` y proporciona una funcionalidad adicional. La clase `CharsetEncoder` representa a un objeto codificador para transformar caracteres Unicode en bytes de un conjunto de caracteres específico. Declara el método `encode` y proporciona una funcionalidad adicional.

Los conjuntos de caracteres solamente pueden decodificarse a, y codificarse a partir de, Unicode. Para realizar conversiones entre dos conjuntos de caracteres que no sean Unicode, se requieren dos pasos: decodificar de un conjunto de caracteres a Unicode y luego codificar de Unicode a otro conjunto de caracteres.

Ejemplos

En esta sección presentaremos un ejemplo que utiliza las capacidades de NIO. El ejemplo (figura 17.23) demuestra cómo leer y escribir datos en un objeto `FileChannel`, utilizando búferes. Con las nuevas APIs de E/S, cada flujo tiene un canal subyacente asociado. Como no podemos instanciar un objeto `FileChannel` directamente, debemos invocar al método `getChannel` (declarado en las clases `FileOutputStream`, `FileInputStream` y `RandomAccessFile`).

En la figura 17.23 se utilizan objetos `ByteBuffer` para leer y escribir `datos` en un objeto `FileChannel`. El constructor (líneas 11 a 22) crea un archivo `RandomAccessFile` y obtiene el canal del archivo invocando al método `getChannel` de `RandomAccessFile`. La operación de escritura se realiza en el método `escribirAlArchivo` (líneas 25 a 38). En la línea 28 se crea un objeto `ByteBuffer` con capacidad de 14, invocando al método `static allocate` de `ByteBuffer`. En la línea 31 se invoca al método `putInt` de `ByteBuffer` para escribir el entero `100` en la posición actual del búfer. Después de la escritura, la posición actual del búfer se incrementa en cuatro. En la línea 32 se invoca al método `putChar` de `ByteBuffer` para escribir el carácter `'A'` en la posición actual del búfer. Después de la escritura, la posición actual del búfer se incrementa en dos. En la línea 33 se invoca al método `putDouble` de `ByteBuffer` para escribir el valor `double 12.34` en la posición actual del búfer. Después de la escritura, la posición actual del búfer se incrementa en ocho. Para escribir el búfer en el canal, en la línea 36 se invoca el método `flip` de `ByteBuffer` para establecer el límite en la posición actual y la posición en cero, y en la línea 37 se invoca el método `write` de `FileChannel` para escribir bytes en el canal del archivo. El método `write` toma un objeto `ByteBuffer`, escribe en el canal los bytes contenidos en el búfer y devuelve el número de bytes escritos. La posición del archivo se actualiza en la posición que tenía antes de la operación de escritura, más el número de bytes escritos.

```
1  // Fig. 17.23: PruebaFileChannel.java
2  // Demuestra el uso de FileChannel y ByteBuffer.
3  import java.io.*;
4  import java.nio.*;
```

Figura 17.23 Escribir búferes en, y leer búferes de, un objeto `FileChannel`. (Parte 1 de 3.)


```
5  import java.nio.channels.*;
6
7  public class PruebaFileChannel {
8     private FileChannel canalArchivo;
9
10    // constructor sin argumentos
11    public PruebaFileChannel()
12    {
13       // crear archivo de acceso aleatorio y obtener canal del archivo
14       try {
15          RandomAccessFile archivo = new RandomAccessFile( "Prueba", "rw" );
16          canalArchivo = archivo.getChannel();
17       }
18       catch ( IOException excepcionES ) {
19          excepcionES.printStackTrace();
20       }
21
22    } // fin del constructor de PruebaFileChannel
23
24    // escribir en canalArchivo
25    public void escribirAlArchivo() throws IOException
26    {
27       // crear búfer para escritura
28       ByteBuffer bufer = ByteBuffer.allocate( 14 );
29
30       // escribir un int, un char y un double en búfer
31       bufer.putInt( 100 );
32       bufer.putChar( 'A' );
33       bufer.putDouble( 12.34 );
34
35       // voltear búfer y escribirlo en canalArchivo
36       bufer.flip();
37       canalArchivo.write( bufer );
38    }
39
40    // leer de canalArchivo
41    public void leerDeArchivo() throws IOException
42    {
43       String contenido = "";
44
45       // crear búfer para lectura
46       ByteBuffer bufer = ByteBuffer.allocate( 14 );
47
48       // leer búfer de canalArchivo
49       canalArchivo.position( 0 );
50       canalArchivo.read( bufer );
51
52       // voltear búfer para lectura
53       bufer.flip();
54
55       // obtener contenido
56       contenido = bufer.getInt() + ", " + bufer.getChar() + ", " +
57          bufer.getDouble();
58
59       System.out.println( "El archivo contiene: " + contenido );
```

Figura 17.23 Escribir búferes en, y leer búferes de, un objeto `FileChannel`. (Parte 2 de 3.)

http://librosysolucionarios.net

```
60
61          // cerrar canalArchivo
62          canalArchivo.close();
63
64       } // fin del método leerDeArchivo
65
66       public static void main( String[] args )
67       {
68          PruebaFileChannel aplicacion = new PruebaFileChannel();
69
70          // escribir en el archivo y después leer del archivo
71          try {
72             aplicacion.escribirAlArchivo();
73             aplicacion.leerDeArchivo();
74          }
75          catch ( IOException excepcionES ) {
76             excepcionES.printStackTrace();
77          }
78       }
79
80    } // fin de la clase PruebaFileChannel
```

```
El archivo contiene: 100, A, 12.34
```

Figura 17.23 Escribir búferes en, y leer búferes de, un objeto `FileChannel`. (Parte 3 de 3.)

La operación de lectura se realiza en el método `leerDeArchivo` (líneas 41 a 64). En la línea 46 se crea un búfer con capacidad de 14. Recuerde que después de escribir, el canal se posiciona después del último byte escrito. Para leer desde el principio, en la línea 49 se invoca al método `position` de `FileChannel` para establecer la posición del archivo en 0. Esto es equivalente a llamar al método `seek` de un objeto `RandomAccessFile`. En la línea 50 se invoca al método `read` de `FileChannel` para leer bytes del canal en un búfer de bytes. El método `read` llena su argumento `ByteBuffer` con bytes del canal y devuelve el número de bytes leídos. Después de la lectura, la posición en cada búfer de byte se establece en la posición después del último byte leído. Para leer el búfer desde el principio, en la línea 53 se invoca al método `flip` de `ByteBuffer` para establecer la posición en 0. En las líneas 56 y 57 se invoca el método `getInt` para obtener un entero del búfer de bytes, el método `getChar` para obtener un carácter del búfer de bytes y el método `getDouble` para obtener un valor `double` del búfer de bytes. En la línea 62 se invoca el método `close` de `FileChannel` para cerrar el canal del archivo.

Recursos en la Web para las Nuevas APIs de E/S

En esta sección presentamos varias de las herramientas de las Nuevas APIs de E/S de Java. Para obtener más información sobre estas APIs, visite los siguientes sitios Web:

`http://www.onjava.com/pub/a/onjava/2002/10/02/javanio.html`
Este artículo de la revista *OnJava* enlista diez de las mejores cosas que se pueden hacer con las nuevas APIs de E/S.

`http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html`
Este artículo de la revista *JavaWorld* detalla algunas de las nuevas herramientas de E/S.

`http://developer.java.sun.com/developer/technicalArticles/releases/nio/`
Este artículo habla sobre la nueva funcionalidad de E/S incluida en las nuevas APIs de E/S.

`http://java.sun.com/j2se/1.4.1/docs/api/java/nio/package-summary.html`
Este sitio Web contiene la documentación de las APIs para el paquete `java.nio`.

http://librosysolucionarios.net

RESUMEN

- Todos los elementos de datos procesados por una computadora se reducen en combinaciones de ceros y unos. El elemento más pequeño de datos en una computadora (un bit) puede asumir el valor `0` o `1`.
- Los dígitos, letras y símbolos especiales se llaman caracteres. Al conjunto de todos los caracteres utilizados para escribir programas y representar elementos de datos en una computadora específica se le conoce como el conjunto de caracteres de esa computadora. Todo carácter en el conjunto de caracteres de una computadora se representa como un patrón de `1`s y `0`s. (Los caracteres en Java son caracteres Unicode que se componen de dos bytes.)
- Un campo es un grupo de caracteres (o bytes) que transmiten cierto significado.
- Un registro es un grupo de campos relacionados.
- La clase `File` permite a los programas obtener información acerca de un archivo o directorio.
- Java considera a cada archivo como un flujo secuencial de bytes. El programador debe estructurar un archivo de manera apropiada para cumplir con los requerimientos de una aplicación.
- Cada archivo termina en cierta forma de marcador de fin de archivo, dependiente del equipo.
- Los flujos proporcionan conexiones de comunicación entre programas y archivos, memoria u otros programas, a través de una red.
- Los programas utilizan clases del paquete `java.io` para realizar operaciones de entrada y salida.
- Los archivos se abren creando instancias de las clases de flujo `FileInputStream`, `FileOutputStream`, `RandomAccessFile`, `FileReader` y `FileWriter`.
- `InputStream` y `OutputStream` son clases `abstract` que declaran métodos para realizar operaciones de entrada y salida, respectivamente. Las operaciones de entrada/salida en archivos pueden realizarse con `FileInputStream` (una subclase de `InputStream`) y `FileOutputStream` (una subclase de `OutputStream`).
- Las canalizaciones son conexiones de comunicación sincronizadas entre subprocesos. Una canalización se establece entre dos subprocesos. Un subproceso envía datos a otro, escribiendo a un objeto `PipedOutputStream`. El subproceso de destino lee la información de la canalización mediante un objeto `PipedInputStream`.
- Un objeto `PrintStream` es un flujo de salida basado en texto. `System.out` y `System.err` son flujos `PrintStream`.
- Leer datos en forma de bytes sin ningún formato es un proceso rápido, pero crudo. Por lo general, los programas leen datos como agregados de bytes que forman objetos o valores de tipos primitivos tales como `int` y `double`.
- La interfaz `DataInput` es implementada por las clases `DataInputStream` y `RandomAccessFile`; cada una de ellas lee valores de tipo primitivo de un flujo.
- La interfaz `DataOutput` es implementada por las clases `DataOutputStream` y `RandomAccessFile`; cada una de ellas escribe valores de tipo primitivo a un flujo `OutputStream`.
- El uso de un búfer es una técnica para mejorar el rendimiento de las operaciones de E/S. Con un objeto `BufferedOutputStream`, cada instrucción de salida no produce necesariamente una transferencia física real de datos hacia el dispositivo de salida. En vez de ello, cada operación de salida se dirige hacia una región en memoria conocida como búfer, que es lo suficientemente grande como para almacenar los datos de muchas operaciones de salida. Después, la transferencia real hacia el dispositivo de salida se realiza en una sola operación de salida física extensa cada vez que se llena el búfer. Las operaciones de salida dirigidas hacia el búfer de salida en memoria se conocen a menudo como operaciones lógicas de salida.
- Con un objeto `BufferedInputStream` muchos trozos "lógicos" de datos de un archivo se leen como una sola operación de entrada física extensa y se envían a un búfer de memoria. A medida que un programa solicita cada nuevo trozo de datos, éste se toma del búfer. (A este procedimiento se le conoce algunas veces como operación lógica de entrada.) Cuando el búfer está vacío, se lleva a cabo la siguiente operación física de entrada desde el dispositivo de entrada, para leer el siguiente grupo de trozos "lógicos" de datos. Por lo tanto, el número de operaciones físicas de entrada es pequeño, en comparación con el número de solicitudes de lectura emitidas por el programa.
- Con un objeto `BufferedOutputStream`, se puede forzar a un búfer parcialmente lleno para que envíe su contenido al dispositivo en cualquier momento, mediante una llamada explícita al método `flush`.
- Los objetos flujo pueden envolverse para crear objetos flujo con una funcionalidad combinada, tal como escribir objetos en un archivo.
- La interfaz `ObjectInput` es similar a la interfaz `DataInput`, sólo que incluye métodos adicionales para leer objetos `Object` de flujos `InputStream`.

- La interfaz `ObjectOutput` es similar a la interfaz `DataOutput`, sólo que incluye métodos adicionales para escribir objetos `Object` en flujos `InputStream`.
- Las clases `ObjectInputStream` y `ObjectOutputStream` implementan a las interfaces `ObjectInput` y `ObjectOutput`, respectivamente.
- Un objeto `RandomAccessFile` es útil para las aplicaciones de acceso directo, como las aplicaciones para procesar transacciones, sistemas de reservaciones de aerolíneas y sistemas de punto de venta.
- Con un archivo de acceso secuencial, cada solicitud de entrada/salida sucesiva lee o escribe el siguiente conjunto consecutivo de datos en el archivo.
- Con un archivo de acceso aleatorio, cada solicitud de entrada/salida sucesiva podría dirigirse a cualquier parte del archivo; tal vez una sección que esté ampliamente separada de la parte del archivo a la que se hizo referencia en la solicitud anterior.
- Un objeto `ByteArrayInputStream` lee su entrada de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` escribe en un arreglo `byte` en memoria.
- Un objeto `StringBufferInputStream` (una subclase de la clase `abstract InputStream`) recibe su entrada de un objeto `StringBuffer`.
- Un objeto `SequenceInputStream` permite la concatenación de varios objetos `InputStream` de manera que el programa vea al grupo como un flujo `InputStream` continuo. Cuando el programa llega al final de cada flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.
- Las clases `BufferedReader` y `BufferedWriter` permiten el uso eficiente de búferes para los flujos basados en caracteres. Las clases `CharArrayReader` y `CharArrayWriter` leen y escriben, respectivamente, un flujo de caracteres en un arreglo de caracteres. Las clases `FileReader` y `FileWriter` leen y escriben, respectivamente, caracteres en un archivo. Las clases `PipedReader` y `PipedWriter` son flujos de caracteres canalizados. Las clases `StringReader` and `StringWriter` leen y escriben, respectivamente, caracteres en cadenas. Un objeto `PrintWriter` escribe caracteres en un flujo.
- Los archivos se abren en modo de salida creando un objeto de la clase `FileOutputStream`. Se pasa un argumento al constructor: el nombre del archivo. Los archivos existentes se truncan, por lo que se pierden todos los datos en el archivo. Los archivos que no existen son creados.
- Un programa puede procesar ninguno, uno o varios archivos. Cada archivo tiene un nombre único y está asociado con un objeto flujo de archivo apropiado. Todos los métodos de procesamiento de archivos deben hacer referencia a un archivo con el objeto apropiado.
- Un apuntador de posición de archivo indica la posición en el archivo a partir de la cual ocurrirá la siguiente entrada, o en donde se va a colocar la siguiente salida.
- Una manera conveniente de implementar los archivos de acceso aleatorio es utilizando solamente registros de longitud fija. Utilizando esta técnica, un programa puede calcular rápidamente la ubicación exacta de un registro, de manera relativa al principio del archivo. Pueden insertarse datos en un archivo de acceso aleatorio sin necesidad de destruir los demás datos en el archivo. Los datos también pueden actualizarse o eliminarse sin necesidad de volver a escribir todo el archivo.
- Muchos búferes almacenan sus datos en un arreglo de respaldo.
- Cada búfer tiene cuatro propiedades fundamentales: capacidad, límite, posición y marca.
- La capacidad del búfer indica cuántos datos puede almacenar.
- El límite de un búfer es la parte donde terminan los datos válidos. No pueden leerse ni escribirse elementos más allá de ese valor.
- La posición de un búfer indica el siguiente elemento que se leerá o escribirá. Este valor cambia cada vez que se lee o se escribe un elemento en un búfer.
- La marca es un lugar que se recuerda en el búfer. El valor de la marca es indefinido cuando se crea el búfer, pero puede establecerse después.
- El búfer más simple es `ByteBuffer`. Un `ByteBuffer` no directo está respaldado por un arreglo. Con un `ByteBuffer` directo, la JVM trata de utilizar operaciones de E/S nativas del sistema operativo.
- También existen búferes para cada uno de los tipos primitivos, excepto `boolean`. Están las clases `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` y `ShortBuffer`.
- Los objetos `MappedByteBuffer` son un tipo de búfer directo, el cual se asigna directamente a una región de un archivo.
- Los datos se escriben en un búfer mediante una operación colocar (put), y se leen de un búfer mediante una operación obtener (get).

- Una operación obtener o colocar relativa empieza con el elemento en la posición del búfer, y la posición se actualiza en el elemento que se encuentra después del último elemento que se haya leído o escrito.
- Una operación obtener o colocar absoluta toma un índice, que es el elemento inicial para esa operación. Una operación obtener o colocar absoluta no actualiza la posición del búfer.
- Una operación borrar (clear) establece el estado del búfer a la configuración original. El límite del búfer se establece en el valor de su capacidad, y la posición del búfer se establece en 0.
- Una operación voltear (flip) establece el límite en la posición actual y establece la posición en 0.
- Una operación regresar (rewind) establece la posición en 0, sin alterar el límite.
- Una operación restablecer (reset) establece la posición en la marca.
- Un canal es una conexión a un dispositivo de E/S, el cual está diseñado para interactuar de manera eficiente con los búferes. Un canal se abre automáticamente al crearse, y no puede volver a abrirse una vez cerrado.
- La interfaz `ReadableByteChannel` declara el método `read`; la interfaz `WritableByteChannel` declara el método `write`. La interfaz `ByteChannel` es una interfaz de conveniencia que extiende tanto a `ReadableByteChannel` como a `WritableByteChannel`.
- El método `read` toma un argumento `ByteBuffer`, el cual se rellena con bytes provenientes del canal.
- El método `write` toma un objeto `ByteBuffer`, cuyo contenido se escribe en el canal.
- La interfaz `ScatteringByteChannel` implementa a `ReadableByteChannel` y permite lecturas esparcidas. Una lectura esparcida es cuando se llenan varios búferes con una sola operación de lectura.
- La interfaz `GatheringByteChannel` implementa a `WritableByteChannel` y permite escrituras esparcidas. Una escritura esparcida toma datos de varios búferes y los escribe en el canal, en una sola operación.
- La clase `FileChannel` representa un canal conectado a un archivo.
- Un bloqueo de archivo restringe el acceso a una porción de un archivo. La posición es el byte inicial que pertenece al bloqueo, y el tamaño es el número de bytes que pertenecen al bloqueo.
- Un bloqueo excluyente no permite que se creen bloqueos traslapados en su porción del archivo.
- Los bloqueos compartidos pueden traslapar porciones de un archivo, pero no permiten que un bloqueo excluyente se traslape en esa porción.
- El método `lock` es una operación bloqueadora. Esto significa que el método regresará solamente cuando se haya adquirido el bloqueo.
- El método `tryLock` es una operación no bloqueadora. Este método regresa inmediatamente cuando es llamado, pero no garantiza que se haya adquirido el bloqueo.
- Un conjunto de caracteres asigna una secuencia de bytes a una secuencia de caracteres.
- Un objeto de la clase `CharSet` representa a uno de los conjuntos de caracteres.
- El método `decode` convierte los bytes en el conjunto de caracteres (introducido como un objeto `ByteBuffer`) en un objeto `CharBuffer` que contiene caracteres Unicode.
- Dos métodos `encode` convierten a un objeto `CharBuffer` o a una cadena de caracteres Unicode en bytes, en el objeto `Charset` (se envía de salida como un objeto `ByteBuffer`).
- La clase `CharsetDecoder` representa a un objeto decodificador para transformar un conjunto de caracteres específico en caracteres Unicode.
- La clase `CharsetEncoder` representa a un objeto codificador para transformar caracteres Unicode en un conjunto de caracteres específico.
- Para realizar conversiones entre dos conjuntos de caracteres que no sean Unicode, se requieren dos pasos: codificar de un conjunto de caracteres a Unicode y luego decodificar de Unicode a otro conjunto de caracteres.

TERMINOLOGÍA

"`r`", modo de apertura de archivo
"`rw`", modo de apertura de archivo
abrir un archivo
aplicaciones de acceso directo
apuntador de posición de archivo
archivo
archivo de acceso aleatorio
archivo de acceso secuencial
arreglo de respaldo
base de datos
bit (abreviación de dígito binario)
`bloqueo`
bloqueo compartido
bloqueo de archivo
bloqueo excluyente
bloqueos traslapados
búfer
búfer de memoria

http://librosysolucionarios.net

búfer directo
búfer parcialmente lleno
`BufferedInputStream`, clase
`BufferedOutputStream`, clase
`BufferedReader`, clase
`BufferedWriter`, clase
byte
`ByteArrayInputStream`, clase
`ByteArrayOutputStream`, clase
`ByteBuffer`
`ByteChannel`
campo
campo alfabético
campo numérico
canalización
`CANCEL_OPTION`, constante
`canRead`, método de la clase `File`
`canWrite`, método de la clase `File`
capacidad
cerrar el archivo
`Channel`
`CharArrayReader`, clase
`CharArrayWriter`, clase
`CharBuffer`
`Charset`
`CharsetDecoder`
`CharsetEncoder`
clave de registro
`close`, método
conjunto codificado de caracteres
conjunto de caracteres
Conjunto de caracteres Unicode
cuadro de diálogo modal
`DataInput`, interfaz
`DataOutput`, interfaz
datos persistentes
`decode`
dígitos decimales
directorio
`DoubleBuffer`
`encode`
`EndOfFileException`
envoltura de objetos flujo
escritura esparcida
esquema de codificación de caracteres
`exists`, método de la clase `File`
`File`, clase
`FileChannel`
`FileInputStream`, clase
`FileOutputStream`, clase
`FileReader`, clase
`FileWriter`, clase
`FilterInputStream`, clase
`FilterOutputStream`, clase
fin de archivo
`FloatBuffer`
flujo de entrada
flujo de salida
flujo estándar de salida
`GatheringByteChannel`
`getAbsolutePath`, método de la clase `File`
`getChannel`
`getName`, método de la clase `File`
`getParent`, método de la clase `File`
`getPath`, método de la clase `File`
`getSelectedFile`, método
`InputStream`, clase
`InputStreamReader`, clase
`IntBuffer`
`IOException`
`isAbsolute`, método de la clase `File`
`isDirectory`, método de la clase `File`
`isFile`, método de la clase `File`
`java.nio`
`java.nio.charset`
jerarquía de datos
`JFileChooser`, clase
`lastModified`, método de la clase `File`
lectura esparcida
`length`, método de la clase `File`
límite
`list`, método de la clase `File`
`LongBuffer`
`MappedByteBuffer`
marca
marcador de fin de archivo
nombre de archivo
`ObjectInput`, interfaz
`ObjectInputStream`, clase
`ObjectOutput`, interfaz
`ObjectOutputStream`, clase
operación bloqueadora
operación borrar (clear)
operación *colocar (put)* absoluta
operación *colocar (put)* relativa
operación lógica de entrada
operaciones lógicas de salida
operación *obtener (get)* absoluta
operación *obtener (get)* relativa
operación regresar (rewind)
operación restablecer (reset)
operación voltear (flip)
`OutputStream`, clase
`OutputStreamWriter`, clase
`PipedInputStream`, clase
`PipedOutputStream`, clase
`PipedReader`, clase
`PipedWriter`, clase
posición
`PrintStream`, clase
`PrintWriter`, clase
programa para procesar transacciones
`RandomAccessFile`, clase
`read`, método
`ReadableByteChannel`
`readBoolean`, método
`readByte`, método
`readChar`, método
`readDouble`, método
`Reader`, clase
`readFloat`, método
`readFully`, método
`readInt`, método
`readLong`, método
`readObject`, método
`readShort`, método
`readUnsignedByte`, método
`readUnsignedShort`, método
registro
ruta absoluta
ruta relativa
`ScatteringByteChannel`
`seek`, método
`SequenceInputStream`, clase
`Serializable`, interfaz
`setFileSelectionMode`, método
`ShortBuffer`
`showOpenDialog`, método
`showSaveDialog`, método
sistema de administración de bases de datos (DBMS)
`StringReader`, clase
`StringWriter`, clase
`System.err` (flujo de error estándar)
`System.in` (flujo de entrada estándar)
`System.out` (flujo de salida estándar)
truncar un archivo existente
`tryLock`
uso de un búfer
`write`, método
`WritableByteChannel`
`writeBoolean`, método
`writeByte`, método
`writeBytes`, método
`writeChar`, método
`writeChars`, método
`writeDouble`, método
`writeFloat`, método
`writeInt`, método
`writeLong`, método
`writeObject`, método
`Writer`, clase
`writeShort`, método

EJERCICIOS DE AUTOEVALUACIÓN

17.1 Complete las siguientes oraciones:

a) Básicamente, todos los elementos de datos procesados por una computadora se reducen en combinaciones de _____ y _____.
b) El elemento de datos más pequeño que puede procesar una computador se conoce como _____.
c) Un _____ es un grupo de caracteres relacionados.
d) Los dígitos, letras y símbolos especiales se conocen como _____.
e) A un grupo de archivos relacionados se le conoce como _____.
f) El método _____ de las clases de flujo de archivo `FileOutputStream`, `FileInputStream` y `RandomAccessFile` cierra un archivo.
g) El método _____ de `RandomAccessFile` lee un entero del flujo especificado.
h) El método _____ de `RandomAccessFile` lee una línea de texto del flujo especificado.
i) El método _____ de `RandomAccessFile` establece el apuntador de posición de archivo en una ubicación específica de un archivo, para realizar una operación de entrada o salida.

17.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) El programador debe crear explícitamente a `System.in`, `System.out` y `System.err`.
b) Si el apuntador de posición de archivo apunta a una posición en un archivo secuencial que no sea el principio, el archivo debe cerrarse y volverse a abrir para leer desde el principio.
c) No es necesario buscar en todos los registros de un archivo de acceso aleatorio para encontrar un registro.
d) Los registros en los archivos de acceso aleatorio deben tener una longitud uniforme.
e) El método `seek` debe buscar en forma relativa al principio de un archivo.

17.3 Complete las siguientes tareas, suponga que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo `"antmaest.dat"` en modo de entrada; use el objeto `ObjectInputStream` llamado `entAntMaestro` para envolver un objeto `FileInputStream`.
b) Escriba una instrucción que abra el archivo `"trans.dat"` en modo de entrada; use el objeto `ObjectInputStream` llamado `entTransaccion` para envolver un objeto `FileInputStream`.
c) Escriba una instrucción para abrir el archivo `"nuevomaest.dat"` en modo de salida (y creación); use el objeto `ObjectOutputStream` llamado `salNuevoMaestro` para envolver un objeto `FileOutputStream`.
d) Escriba una instrucción que lea un registro del archivo `"antmaest.dat"`. El registro es un objeto de la clase `RegistroCuentas`; use el objeto `ObjectInputStream` llamado `entAntMaestro`.
e) Escriba una instrucción que lea un registro del archivo `"trans.dat"`. El registro es un objeto de la clase `RegistroTransacciones`; use el objeto `ObjectInputStream` llamado `entTransaccion`.
f) Escriba una instrucción que escriba un registro en el archivo `"nuevomaest.dat"`. El registro es un objeto de tipo `RegistroCuentas`; use el objeto `ObjectInputStream` llamado `salNuevoMaestro`.

17.4 Encuentre el error en cada uno de los siguientes bloques de código y muestre cómo corregirlo.

a) Suponga que se declaran `cuenta`, `compania` y `monto`.

```
ObjectOutputStream flujoSalida;

flujoSalida.writeInt( cuenta );
flujoSalida.writeChars( compania );
flujoSalida.writeDouble( monto );
```

b) Las siguientes instrucciones deben leer un registro del archivo `"porpagar.dat"`. El objeto `entPorPagar` de `ObjectInputStream` hace referencia a este archivo, y el objeto `entPorCobrar` de `FileInputStream` hace referencia al archivo `"porcobrar.dat"`.

```
cuenta = entPorCobrar.readInt();
companiaID = entPorCobrar.readLong();
monto = entPorCobrar.readDouble();
```

http://librosysolucionarios.net

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

17.1 a) Unos, ceros. b) Bit. c) Archivo. d) Caracteres. e) Base de datos. f) `close`. g) `readInt`. h) `readLine`. i) `seek`.

17.2 a) Falso. Estos tres flujos se crean automáticamente para el programador.
b) Verdadero.
c) Verdadero.
d) Falso. Los registros en un archivo de acceso aleatorio tienen normalmente una longitud uniforme.
e) Verdadero.

17.3 a) `ObjectInputStream entAntMaestro = new ObjectInputStream( new FileInputStream ( "antmaest.dat" ) );`
b) `ObjectInputStream entTransaccion = new ObjectInputStream( new FileInputStream( "trans.dat" ) );`
c) `ObjectOutputStream salNuevoMaestro = new ObjectOutputStream( new FileOutputStream( "nuevomaest.dat" ) );`
d) `registroCuentas = ( RegistroCuentas ) entAntMaestro.getObject();`
e) `registroTransacciones = (RegistroTransacciones) entTransaccion.getObject();`
f) `salNuevoMaestro.writeObject( nuevoRegistroCuentas );`

17.4 a) Error: El archivo no se ha abierto antes de tratar de escribir datos en el flujo.
Corrección: Crear un nuevo objeto `ObjectOutputStream` que envuelva a un objeto `FileOutputStream`, para poder abrir el archivo en modo de salida.
b) Error: Se está utilizando el objeto `FileInputStream` incorrecto para leer del archivo `"porpagar.dat"`.
Corrección: Usar el objeto `entPorPagar` para hacer referencia a `"porpagar.dat"`.

EJERCICIOS

17.5 Llene los espacios en blanco en cada uno de los siguientes enunciados:
a) La computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundario, tales como _____.
b) Un _____ está compuesto de varios campos.
c) Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse un campo en cada registro como _____.
d) La mayor parte de la información en los sistemas computacionales se almacena en archivos _____.
e) Los objetos flujo estándar son _____, _____ y _____.

17.6 Determine cuál de los siguientes enunciados es *verdadero* y cuál es *falso*. Si es *falso*, explique por qué.
a) Las impresionantes funciones realizadas por las computadoras involucran esencialmente la manipulación de ceros y unos.
b) Las personas especifican los programas y elementos de datos como caracteres; después, las computadoras manipulan y procesan estos caracteres como grupos de ceros y unos.
c) El código postal de cinco dígitos de una persona es un ejemplo de un campo numérico.
d) La dirección en donde vive una persona se considera generalmente como un campo alfabético.
e) Los elementos de datos que se representan en las computadoras forman una jerarquía de datos, en la cual los elementos de datos se hacen más grandes y complejos, a medida que progresamos de campos a caracteres, de caracteres a bits, etcétera.
f) Una clave de registro identifica que un registro pertenece a un campo específico.
g) Las compañías almacenan toda su información en un solo archivo, para poder facilitar el procesamiento computacional de la información. Cuando un programa crea un archivo, éste es retenido automáticamente por la computadora para cuando se haga referencia a él en un futuro.

17.7 El ejercicio de autoevaluación 17.3 pide al lector que escriba una serie de instrucciones individuales. En realidad, estas instrucciones forman el núcleo de un tipo importante de programa para procesar archivos: un programa para asociar archivos. En el procesamiento de datos comercial, es común tener varios archivos en cada sistema de aplicaciones. Por ejemplo, en un sistema de cuentas por cobrar hay generalmente un archivo maestro, el cual contiene información detallada acerca de cada cliente, como su nombre, dirección, número telefónico, saldo deudor, límite de crédito, términos de descuento, acuerdos contractuales y posiblemente un historial condensado de compras recientes y pagos en efectivo.
a) A medida que ocurren las transacciones (es decir, a medida que se generan las ventas a clientes, los pagos por correo), la información acerca de ellas se introduce en un archivo. Al final de cada periodo de negocios (por ejemplo, un mes para algunas compañías, una semana para otras y un día en algunos casos), el archivo de transacciones (llamado `"trans.dat"` en el ejercicio de autoevaluación 17.3) se aplica al archivo maestro (llamado `"antmaest.dat"` en el ejercicio de autoevaluación 17.3) para actualizar el registro de compras y pagos de cada cuenta. Durante una actualización, el archivo maestro se reescribe como el archivo `"nuevomaest.dat"`, el cual se utiliza al final del siguiente periodo de negocios para empezar de nuevo el proceso de actualización.
b) Los programas para asociar archivos deben tratar con ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, una asociación no siempre ocurre. Tal vez un cliente en el archivo maestro no tenga compras ni pagos en efectivo en el periodo actual de negocios; por lo tanto, no aparecerá ningún registro para este cliente en el archivo de transacciones. De manera similar, un cliente que haya realizado compras o pagos en efectivo podría haberse mudado recientemente a esta comunidad, y tal vez la compañía no haya tenido la oportunidad de crear un registro maestro para este cliente.
c) Utilice las instrucciones en el ejercicio de autoevaluación 17.3 como una base para escribir un programa completo para asociar archivos de cuentas por cobrar. Utilice el número de cuenta en cada archivo como la clave de registro para fines de asociar los archivos. Suponga que cada archivo es un archivo secuencial con registros almacenados en orden ascendente, por número de cuenta.
d) Cuando ocurra una asociación (es decir, que aparezcan registros con el mismo número de cuenta tanto en el archivo maestro como en el archivo de transacción), agregue la cantidad en dólares del archivo de transacciones al balance actual en el archivo maestro, y escriba el registro `"nuevomaest.dat"`. (Suponga que las compras se indican mediante montos positivos en el archivo de transacciones, y los pagos mediante montos negativos.) Cuado hay un registro maestro para una cuenta específica, pero no hay un registro de transacciones correspondiente, simplemente escriba el registro maestro en `"nuevomaest.dat"`. Cuando haya un registro de transacciones pero no un registro maestro correspondiente, imprima en un archivo de registro el mensaje `"Hay un registro de transacciones no asociado para el número de cuenta ..."` (utilice el número de cuenta del registro de transacciones).
e) El archivo de registro debe ser un archivo de texto llamado `"registro.txt"`. Este archivo debe crearse utilizando el objeto `PrintStream` llamado `archRegistro`, para envolver un objeto `FileOutputStream`. Pueden escribirse mensajes de error en este archivo, utilizando el método `println` de la clase `PrintStream`.

17.8 Después de escribir el programa del ejercicio 17.7, escriba un programa simple para crear datos de prueba y comprobar la ejecución del programa. Utilice como ejemplo los datos de las cuentas que vienen en las figuras 17.24 y 17.25. Ejecute el programa del ejercicio 17.7, utilizando los archivos de los datos de prueba creados en este ejercicio. Imprima el nuevo archivo maestro. Compruebe que las cuentas se hayan actualizado correctamente.

Archivo maestro Número de cuenta	Nombre	Balance
100	Adrián González	348.17
300	María Martínez	27.19
500	Samuel Contreras	0.00
700	Susana Cabrera	−14.22

Figura 17.24 Datos de ejemplo para el archivo maestro.

Archivo de transacciones Número de cuenta	Monto de la transacción
100	27.14
300	62.11
400	100.56
900	82.17

Figura 17.25 Datos de ejemplo para el archivo de transacciones.

http://librosysolucionarios.net

Número de cuenta	Monto en dólares
300	83.89
700	80.78
700	1.53

Figura 17.26 Registros de transacciones adicionales.

17.9 Es posible (y muy común) tener varios registros de transacciones con la misma clave de registro. Esta situación ocurre cuando un cliente hace varias compras y pagos en efectivo durante un periodo de negocios. Reescriba su programa para asociar archivos de cuentas por cobrar del ejercicio 17.7, para proporcionar la posibilidad de manejar varios registros de transacciones con la misma clave de registro. Modifique los datos de prueba del ejercicio 17.8 para incluir los registros de transacciones adicionales de la figura 17.26.

17.10 Usted es el dueño de una ferretería y necesita llevar un inventario que le pueda decir cuáles son las distintas herramientas que tiene, cuántas tiene en existencia de cada una y su costo. Escriba un programa que inicialice el archivo de acceso aleatorio `"ferreteria.dat"` con 100 registros vacíos, que le permita introducir los datos correspondientes a cada herramienta, enlistar todas sus herramientas, eliminar un registro de una herramienta que ya no tenga en existencia y que le permita actualizar *cualquier* información en el archivo. El número de identificación de herramienta deberá ser el número de registro. Use la información en la figura 17.27 para empezar su archivo.

17.11 *(Generador de palabras de números telefónicos.)* Los teclados telefónicos estándar contienen los dígitos del cero al nueve. Cada uno de los números del dos al nueve tienen tres letras asociadas. (Vea la figura 17.28.) A muchas personas se les dificulta memorizar números telefónicos, por lo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico sea 686-3767 podría utilizar la correspondencia indicada en la figura 17.28 para desarrollar la palabra de siete letras "NUMEROS". Cada palabra de siete letras corresponde exactamente a un número telefónico de siete dígitos. El restaurante que desea incrementar su negocio de comidas para llevar podría lograrlo utilizando el número 266-4327 (es decir, "COMIDAS").

Registro #	Nombre de herramienta	Cantidad	Costo
3	Lijadora eléctrica	18	35.99
19	Martillo	128	10.00
26	Sierra caladora	16	14.25
39	Podadora de césped	10	79.50
56	Sierra eléctrica	8	89.99
76	Destornillador	236	4.99
81	Mazo	32	19.75
88	Llave española	65	6.48

Figura 17.27 Datos para el ejercicio 17.10.

Dígito	Letras
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	W X Y

Figura 17.28 Dígitos y letras de los teclados telefónicos.

Cada número telefónico de siete letras corresponde a muchas palabras de siete letras separadas. Desafortunadamente, la mayoría de estas palabras representan yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una carpintería se complazca en saber que el número telefónico de su talles, 683-2537, corresponde a "MUEBLES" El propietario de una tienda de licores estaría, sin duda, feliz de averiguar que el número telefónico 232-4327 corresponde a "BEBIDAS". Un veterinario con el número telefónico 627-2682 corresponde a las letras "MASCOTA". El propietario de una tienda de música estaría complacido en saber que su número telefónico 687-4225 corresponde a "MUSICAL".

Escriba un programa que, dado un número de siete dígitos, utilice un objeto `PrintStream` para escribir en un archivo todas las combinaciones posibles de palabras de siete letras que corresponden a ese número. Hay 2187 (3^7) combinaciones posibles. Evite los números telefónicos con los dígitos 0 y 1.

17.12 La figura 7.8 contiene un arreglo de respuestas a una encuesta, el cual está codificado directamente en el programa. Suponga que deseamos procesar resultados de encuestas que se guarden en un archivo. Este ejercicio requiere de dos programas separados. Primero, cree una aplicación que pida al usuario las respuestas de la encuesta y que escriba cada respuesta en un archivo. Utilice un objeto `DataOutputStream` que envuelva a un objeto `FileOutputStream` para crear un archivo llamado `numeros.dat`. Cada entero debe escribirse utilizando el método `writeInt` de `DataOutputStream`. Después modifique el programa de la figura 7.8 para leer las respuestas de la encuesta del archivo `numeros.dat`. Las respuestas deben leerse del archivo mediante el uso de un objeto `DataInputStream` envolviendo a un objeto `FileInputStream`. Deberá utilizar el método `readInt` de la clase `DataInputStream` para introducir un entero del archivo a la vez. El programa deberá seguir leyendo respuestas hasta que llegue al fin del archivo, punto en el cual el método `readInt` lanzará una excepción `EOFException`. Los resultados deberán escribirse en un archivo de texto y mostrarse en una ventana. Use un objeto `PrintWriter` que envuelva a un objeto `FileWriter` para escribir en el archivo `"salida.txt"`. Use el método `print` de la clase `PrintWriter` para escribir una cadena en el archivo.

17.13 Modifique el ejercicio 14.17 para permitir al usuario escribir el arreglo de objetos `MiFigura` en un archivo, y para leerlo de un archivo. Use un objeto `ObjectOutputStream` que envuelva a un objeto `FileOutputStream` para escribir en el archivo, y un objeto `ObjectInputStream` que envuelva a un objeto `FileInputStream` para leer del archivo. Escriba el número de figuras almacenadas actualmente en el arreglo, mediante el método `writeInt`. Después escriba el arreglo de figuras. El entero se utilizará para llevar el registro de la siguiente posición disponible en el arreglo, de manera que puedan agregarse más figuras cuando el arreglo se lea del archivo. Cuando vaya a leer datos del archivo, lea el entero con el método `readInt`. Observe que el entero y el arreglo deben leerse en el mismo orden en el que se escribieron.

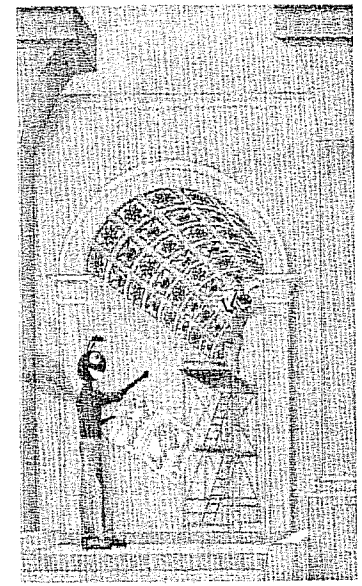
http://librosysolucionarios.net

18

Redes

Objetivos

- Comprender el trabajo en red en Java con URLs, sockets y datagramas.
- Implementar aplicaciones de red en Java, mediante el uso de sockets y datagramas.
- Aprender a implementar clientes y servidores en Java, que se comuniquen entre sí.
- Saber cómo implementar aplicaciones de colaboración basada en red.
- Construir un servidor con subprocesamiento múltiple.



Si la presencia de electricidad puede hacerse visible en cualquier parte de un circuito, no veo la razón por la cual la inteligencia no pueda transmitirse instantáneamente mediante la electricidad.
Samuel F. B. Morse

Sr. Watson, venga aquí, quiero verlo.
Alexander Graham Bell

Lo que las redes de vías de ferrocarril, carreteras y canales fueron en otra época, las redes de telecomunicaciones, información y computación lo son hoy.
Bruno Kreisky

La ciencia tal vez nunca desarrolle un mejor sistema de comunicación para la oficina que el "coffee break".
Earl Wilson

Actualmente es un problema accesar a los gigabits por medio de "punybaud".
J. C. R. Licklider

http://librosysolucionarios.net

Plan general

18.1 Introducción
18.2 Manipulación de URLs
18.3 Cómo leer un archivo en un servidor Web
18.4 Cómo establecer un servidor simple utilizando sockets de flujo
18.5 Cómo establecer un cliente simple utilizando sockets de flujo
18.6 Interacción entre cliente/servidor mediante conexiones de socket de flujo
18.7 Interacción entre cliente/servidor sin conexión mediante datagramas
18.8 Juego de tres en raya (gato) tipo cliente/servidor, utilizando un servidor con subprocesamiento múltiple
18.9 La seguridad y la red
18.10 Servidor y cliente para conversaciones DeitelMessenger
18.10.1 `ServidorDeitelMessenger` y las clases de soporte
18.10.2 Cliente `DeitelMessenger` y las clases de soporte
18.11 Generalidades acerca del trabajo en red con NIO
18.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.io` y `java.net`
18.12.1 Patrones de diseño de creación
18.12.2 Patrones de diseño de estructura
18.12.3 Patrones de arquitectura
18.12.4 Conclusión
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

18.1 Introducción

Hay mucha emoción en cuanto a Internet y World Wide Web. Internet enlaza la "información de todo el mundo". World Wide Web facilita el uso de Internet y le proporciona los beneficios de multimedia. Las organizaciones ven a Internet y a la Web como algo crucial para sus estrategias de sistemas de información. Java proporciona una variedad de herramientas de red integradas, las cuales facilitan el desarrollo de aplicaciones basadas en Internet y Web. Java puede permitir a los programas buscar información en todo el mundo y colaborar con programas que se ejecutan en otras computadoras a nivel internacional, nacional o solamente dentro de una organización. Java puede permitir que los applet (subprogramas) y aplicaciones se comuniquen entre sí (lo cual está sujeto a ciertas restricciones de seguridad).

Las redes son un tema masivo y complejo. Los estudiantes de ciencias computacionales e ingeniería computacional por lo general toman un curso de nivel superior (con duración de un semestre completo) sobre redes de computadoras y continúan estudiando a nivel de graduados. Java proporciona un vasto complemento de herramientas de red, y probablemente se utilice como un vehículo de implementación en cursos de redes computacionales. En *este libro* le presentamos una porción de los conceptos y herramientas de Java para trabajo en red. Para ver herramientas de red más avanzadas, consulte nuestro libro *Advanced Java 2 Platform How to Program*.

En Java, las herramientas de red se agrupan en varios paquetes. Las herramientas de red fundamentales se declaran mediante clases e interfaces del paquete `java.net`, mediante el cual Java ofrece *comunicaciones basadas en flujos* que permiten a las aplicaciones ver las redes como flujos de datos. Las clases e interfaces del paquete `java.net` también ofrecen *comunicaciones basadas en paquetes*, para transmitir *paquetes* individuales de información; esto se utiliza comúnmente para transmitir audio y vídeo a través de Internet. En este capítulo mostraremos cómo crear y manipular sockets, y cómo comunicarnos con los paquetes de datos.

Nuestra discusión sobre las redes se enfoca en ambos lados de una *relación cliente-servidor*. El *cliente* solicita que se realice cierta acción, y el *servidor* realiza la acción y responde al cliente. Una implementación común del modelo de petición-respuesta se da entre los navegadores y servidores Web. Cuando un usuario selecciona un sitio Web para navegar mediante un navegador (la aplicación cliente), se envía una petición al servidor Web apropiado (la aplicación servidor). Por lo general, el servidor responde al cliente enviando una página Web de HTML apropiada.

Presentaremos las *comunicaciones basadas en sockets* en Java, las cuales permiten a las aplicaciones ver las redes como si fueran E/S de archivo; un programa puede leer de un *socket* o escribir en un socket de una manera tan simple como leer o escribir en un archivo. Mostraremos cómo crear y manipular sockets. Java proporciona *sockets de flujo* y *sockets de datagrama*. Con los *sockets de flujo*, un proceso establece una *conexión* con otro proceso. Mientras la conexión está en pie, los datos fluyen entre los procesos en *flujos* continuos. Se dice que los sockets de flujo proporcionan un *servicio orientado a la conexión*. El protocolo utilizado para la transmisión es el popular *TCP (Protocolo de control de transmisión)*.

Con los *sockets de datagrama* se transmiten *paquetes* individuales de información. Éste no es el protocolo adecuado para los programadores cotidianos ya que, a diferencia de TCP, el protocolo utilizado (*UDP*, el *Protocolo de datagramas de usuario*) es un *servicio sin conexión* y no garantiza que los paquetes lleguen en un orden específico. De hecho pueden perderse paquetes, o pueden duplicarse e incluso pueden llegar fuera de secuencia. Por lo tanto, con UDP se requiere de programación adicional por parte del programador para tratar con estos problemas (si el programador así lo quiere). UDP es más apropiado para aplicaciones de red en las que no se requiere la comprobación de errores y la confiabilidad de TCP. Los sockets de flujo y el protocolo TCP son más convenientes para la gran mayoría de programadores de Java.

Tip de rendimiento 18.1

Los servicios sin conexión generalmente ofrecen un mayor rendimiento, pero son menos confiables que los servicios orientados a las conexiones.

Tip de portabilidad 18.1

TCP, UDP y los protocolos relacionados permiten que una gran variedad de sistemas computacionales heterogéneos (es decir, sistemas computacionales con diferentes procesadores y sistemas operativos) se comuniquen unos con otros.

En este capítulo incluimos un ejemplo práctico en el que implementamos una aplicación cliente/servidor para conversar (chat), similar a los populares servicios de mensajería instantánea en Web hoy en día. En este programa se incorporan muchas técnicas de red que introduciremos en este capítulo. El programa también introduce el término *transmisión múltiple (multicasting)*, en donde un servidor puede *publicar* información y los clientes pueden *suscribirse* a esa información. Cada vez que el servidor publica más información, todos los suscriptores la reciben. A lo largo de los ejemplos de este capítulo veremos que muchos de los detalles del trabajo en red son manejados por las clases de Java que utilizamos.

18.2 Manipulación de URLs

Internet ofrece muchos protocolos. El *Protocolo de transferencia de hipertexto (HTTP)* que forma la base de World Wide Web, utiliza *URIs (Identificadores uniformes de recursos)* para identificar datos en Internet. Los URIs que especifican las ubicaciones de documentos se conocen como *URLs (Localizadores uniformes de recursos)*. Los URLs comunes hacen referencia a archivos o directorios y pueden hacer referencia a objetos que realizan tareas complejas, como búsquedas en bases de datos y en Internet. Si usted conoce el URL de documentos HTML que estén públicamente disponibles en Web, puede acceder a esos datos a través de HTTP. Java facilita la manipulación de URLs. Si se utiliza un URL que haga referencia a la ubicación exacta de un recurso (como una página Web) como un argumento para el método `showDocument` de la interfaz `AppletContext`, el navegador en el que se ejecuta el applet mostrará el recurso en el URL especificado. El applet de las figuras 18.1 y 18.2 demuestra el uso de las herramientas simples de red. El applet permite al usuario seleccionar una página Web de un objeto `JList`, y hace que el navegador muestre la página correspondiente. En este ejemplo, el trabajo en red es realizado por el navegador.

Este applet aprovecha los *parámetros de applet* especificados en el documento HTML que invoca al applet. Al navegar por World Wide Web, a menudo se encontrará applet en el dominio público; puede utilizarlos

http://librosysolucionarios.net

```
<html>
<title>Selector de sitios</title>
<body>
   <applet code = "SelectorSitios.class" width = "300" height = "75">
      <param name = "titulo0" value = "Página inicial de Java">
      <param name = "ubicacion0" value = "http://java.sun.com/">
      <param name = "titulo1" value = "Deitel">
      <param name = "ubicacion1" value = "http://www.deitel.com/">
      <param name = "titulo2" value = "JGuru">
      <param name = "ubicacion2" value = "http://www.jGuru.com/">
      <param name = "titulo3" value = "JavaWorld">
      <param name = "ubicacion3" value = "http://www.javaworld.com/">
   </applet>
</body>
</html>
```

Figura 18.1 Documento HTML para cargar el `applet SelectorSitios`.

sin costo en sus propias páginas Web (generalmente a cambio de dar créditos al creador del applet). Una característica común de dichos applet es la habilidad de personalizar el applet mediante los parámetros que se proporcionan desde el archivo HTML que invoca al applet. Por ejemplo, en la figura 18.1 se muestra el HTML que invoca al objeto `SelectorSitios` en la figura 18.2.

El documento HTML contiene ocho parámetros especificados con el *elemento* ***`param`***; estas líneas deben aparecer entre las marcas `applet` inicial y final. El applet puede leer estos valores y utilizarlos para personalizarse a sí mismo. Puede aparecer cualquier número de marcas `param` entre las marcas `applet` inicial y final. Cada parámetro tiene un ***`nombre`*** y un ***`valor`***. El método ***`getParameter`*** de `Applet` recupera el `valor` asociado con un nombre de parámetro específico, y devuelve ese `valor` como una cadena. El argumento que se pasa a `getParameter` es una cadena que contiene el nombre del parámetro en el elemento `param`. En este ejemplo, los parámetros representan el título de cada sitio Web que puede seleccionar el usuario, junto con la ubicación de cada sitio. Puede especificarse cualquier número de parámetros para este applet. Sin embargo, estos parámetros deben nombrarse como `titulo#`, en donde el valor de # empieza en 0 y se incrementa en uno para cada nuevo título. Cada título debe tener un parámetro de ubicación correspondiente, de la forma `ubicacion#`, en donde el valor de # empieza en 0 y se incrementa en uno para cada nueva ubicación. La instrucción:

```
String titulo = getParameter( "titulo0" );
```

obtiene el valor asociado con el parámetro `"titulo0"` y lo asigna a la referencia `titulo`. Si no hay una marca `param` que contenga el parámetro especificado, `getParameter` devuelve `null`.

El applet (figura 18.2) obtiene del documento HTML (figura 18.1) las opciones a mostrar en el objeto `JList` del applet. La clase `SelectorSitios` utiliza un objeto ***`HashMap`*** (paquete ***`java.util`***) para almacenar los nombres de sitios y sus URLs. Un objeto `HashMap` almacena los *pares clave/valor*. El programa utiliza la *clave* para almacenar y recuperar el *valor* asociado en el objeto `HashMap`. En este ejemplo, la *clave* es la cadena en el objeto `JList` que representa el nombre del sitio Web, y el valor es un objeto `URL` que almacena la ubicación del sitio Web que se mostrará en el navegador. La clase `HashMap` proporciona dos métodos de importancia en este ejemplo: ***`put`*** y ***`get`***. El método `put` toma dos argumentos (una clave y su valor asociado) y coloca el valor en el objeto `HashMap`, en una ubicación determinada por la clave. El método `get` toma un argumento (una clave) y devuelve una referencia `Object` del valor correspondiente. La clase `SelectorSitio` también contiene un objeto `Vector` (paquete `java.util`), en el cual se colocan los nombres de los sitios para que puedan utilizarse en la inicialización del objeto `JList` (una versión del constructor de `JList` recibe un objeto `Vector`). Un objeto `Vector` es un arreglo de referencias `Object` que puede cambiar su tamaño en forma dinámica. La clase `Vector` proporciona el método ***`add`*** para agregar un nuevo elemento al final del objeto `Vector`. (Hablaremos detalladamente sobre la clase `Vector` en el capítulo 21, y sobre la clase `HashMap` en el capítulo 22.)

http://librosysolucionarios.net


```
// Fig. 18.2: SelectorSitios.java
// Este programa utiliza un botón para cargar un documento a partir de un URL.
import java.net.*;
import java.util.*;
import java.awt.*;
import java.applet.AppletContext;
import javax.swing.*;
import javax.swing.event.*;

public class SelectorSitios extends JApplet {
   private HashMap sitios;          // nombres y URLs de los sitios
   private Vector nombresSitios;    // nombres de los sitios
   private JList selectorSitios;    // lista de sitios a elegir

   // leer parámetros de HTML y configurar la GUI
   public void init()
   {
      // crear HashMap y Vector
      sitios = new HashMap();
      nombresSitios = new Vector();

      // obtener los parámetros del documento HTML
      obtenerSitiosDeParametrosHTML();

      // crear componentes de la GUI y preparar distribución de la interfaz
      Container contenedor = getContentPane();
      contenedor.add( new JLabel( "Seleccione un sitio para navegar" ),
         BorderLayout.NORTH );

      selectorSitios = new JList( nombresSitios );
      selectorSitios.addListSelectionListener(

         new ListSelectionListener() {

            // ir al sitio que seleccionó el usuario
            public void valueChanged( ListSelectionEvent evento )
            {
               // obtener nombre del sitio seleccionado
               Object objeto = selectorSitios.getSelectedValue();

               // usar nombre del sitio para localizar URL correspondiente
               URL nuevoDocumento = ( URL ) sitios.get( objeto );

               // obtener referencia al contenedor de applet
               AppletContext navegador = getAppletContext();

               // indicar al contenedor de applet para cambiar de página
               navegador.showDocument( nuevoDocumento );
            }

         } // fin de la clase interna

      ); // fin de la llamada a addListSelectionListener

      contenedor.add( new JScrollPane( selectorSitios ),
```

Figura 18.2 Cómo cargar un documento de un URL en un navegador. (Parte 1 de 3.)

```
56            BorderLayout.CENTER );
57
58      } // fin del método init
59
60      // obtener parámetros del documento HTML
61      private void obtenerSitiosDeParametrosHTML()
62      {
63         // buscar parámetros del applet en documento HTML y agregarlos a HashMap
64         String titulo, ubicacion;
65         URL url;
66         int contador = 0;
67
68         titulo = getParameter( "titulo" + contador ); // obtener título del primer sitio
69
70         // iterar hasta que no haya más parámetros en el documento HTML
71         while ( titulo != null ) {
72
73            // obtener ubicación del sitio
74            ubicacion = getParameter( "ubicacion" + contador );
75
76            // colocar título/URL en HashMap y título en Vector
77            try {
78               url = new URL( ubicacion );      // convertir ubicación a URL
79               sitios.put( titulo, url );       // colocar título/URL en HashMap
80               nombresSitios.add( titulo );     // colocar título en Vector
81            }
82
83            // procesar formato incorrecto de URL
84            catch ( MalformedURLException excepcionURL ) {
85              excepcionURL.printStackTrace();
86            }
87
88            ++contador;
89            titulo = getParameter( "titulo" + contador ); // obtener título del siguiente sitio
90
91         } // fin de instrucción while
92
93      } // fin del método obtenerSitiosDeParametrosHTML
94
95   } // fin de la clase SelectorSitios
```

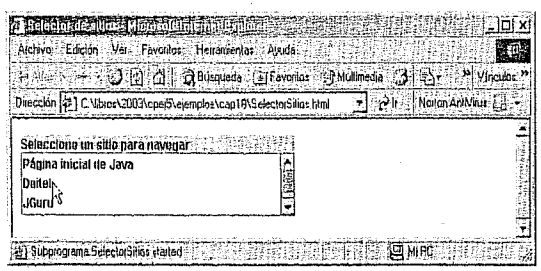


Figura 18.2 Cómo cargar un documento de un URL en un navegador. (Parte 2 de 3.)

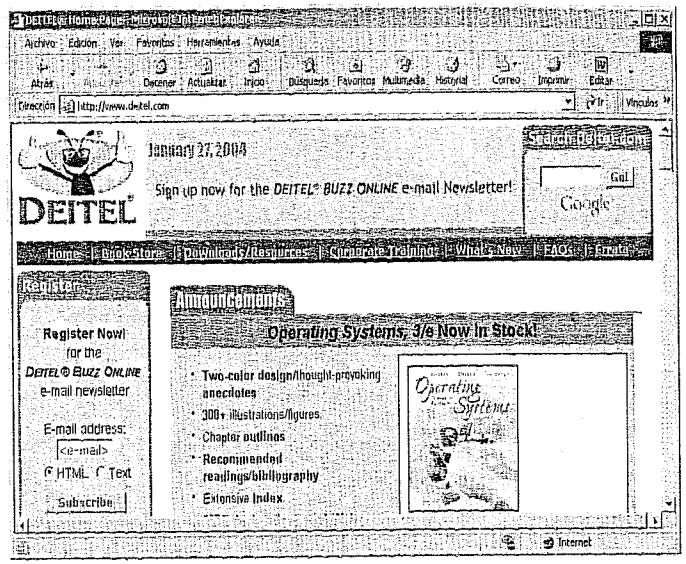


Figura 18.2 Cómo cargar un documento de un URL en un navegador. (Parte 3 de 3.)

En las líneas 19 a 20 del método `init` del applet (líneas 16 a 58) se crea un objeto `HashMap` y un objeto `Vector`. En la línea 23 se hace una llamada a nuestro sitio utilitario `obtenerSitiosDeParametrosHTML` (declarado en las líneas 61 a 93) para obtener los parámetros de HTML del documento HTML que invocó al applet.

En el método `obtenerSitiosDeParametrosHTML`, en la línea 68 se utiliza el método `getParameter` de `Applet` para obtener el título de un sitio Web. Si el `titulo` no es `null`, el ciclo de las líneas 71 a 91 empieza a ejecutarse. En la línea 74 se utiliza el método `getParameter` de `Applet` para obtener la ubicación del sitio Web. En la línea 78 se utiliza la `ubicación` como el valor de un nuevo objeto `URL`. El constructor de `URL` determina si su argumento representa a un URL válido. Si no es así, el constructor de `URL` lanza una excepción `MalformedURLException`. Observe que el constructor de `URL` debe ser llamado en un bloque `try`. Si el constructor de `URL` genera una excepción `MalformedURLException`, la llamada a `printStackTrace` (línea 85) hace que el programa muestre un rastreo de la pila. Después el programa tratará de obtener el título del siguiente sitio Web. El programa no agrega el sitio del URL inválido al objeto `HashMap`, por lo que ese título no se mostrará en el objeto `JList`.

Para un `URL` correcto, en la línea 79 se colocan el `titulo` y el `URL` en el objeto `HashMap`, y en la línea 80 se agrega el `titulo` al objeto `Vector`. En la línea 89 se obtiene el siguiente título del documento HTML. Cuando la llamada a `getParameter` en la línea 89 devuelve `null`, el ciclo termina.

Cuando el método `obtenerSitiosDeParametrosHTML` regresa a `init`, en las líneas 26 a 56 se construye la GUI del applet. En las líneas 27 y 28 se agrega la etiqueta `JLabel` "`Seleccione un sitio para navegar`" a la sección `NORTH` del esquema `BorderLayout` del panel de contenido. En las líneas 31 a 53 se registra un objeto `ListSelectionListener` para manejar los eventos de `selectorSitios`. En las líneas 55 y 56 se agrega `selectorSitios` a la sección `CENTER` del esquema `BorderLayout` del panel de contenido.

Cuando el usuario selecciona uno de los sitios Web listados en `selectorSitios`, el programa llama al método `valueChanged` (líneas 36 a 49). En la línea 39 se obtiene del objeto `JList` el nombre del sitio seleccionado. En la línea 42 se pasa el nombre del sitio seleccionado (la *clave*) al método `get` de `HashMap`, el cual localiza y devuelve una referencia `Object` al objeto `URL` correspondiente (el *valor*). El operador de conversión de `URL` convierte la referencia a un `URL` que puede asignarse a la referencia `nuevoDocumento`.

http://librosysolucionarios.net

En la línea 45 se utiliza el método `getAppletContext` de `Applet` para obtener una referencia a un objeto `AppletContext` que representa el contenedor del applet. En la línea 48 se utiliza la referencia `navegador` para invocar el método `showDocument`, el cual recibe un objeto `URL` como argumento, y lo pasa al objeto `AppletContext` (es decir, el navegador). El navegador muestra en su ventana actual el recurso Web asociado con ese `URL`. En este ejemplo, todos los recursos son documentos HTML.

Para los programadores familiarizados con los *marcos de HTML*, hay una segunda versión del método `showDocument` de `AppletContext` que permite a un applet especificar lo que se conoce como el *marco de destino*, en el cual se mostrará el recurso Web. La segunda versión de `showDocument` toma dos argumentos: un objeto `URL` que especifica el recurso a mostrar, y una cadena que representa el marco de destino. Hay algunos marcos de destino especiales que pueden utilizarse como el segundo argumento. El marco de destino `_blank` ocasiona que se muestre el contenido del URL especificado en una nueva ventana del navegador Web. El marco de destino `_self` especifica que el contenido del URL especificado debe mostrarse en el mismo marco que el applet (la página HTML del applet se reemplaza en este caso). El marco de destino `_top` especifica que el navegador debe eliminar los marcos actuales en la ventana del navegador y después mostrar el contenido del URL especificado en la ventana actual. Para obtener más información sobre HTML y los marcos vea el sitio Web del *Consorcio World Wide Web (W3C)* en la dirección `www.w3.org`.

Tip para prevenir errores 18.1

El `applet` de la figura 18.2 debe ejecutarse desde un navegador Web como Netscape o Microsoft Internet Explorer, para que se pueda ver el resultado de mostrar otra página Web. El `appletviewer` es capaz de ejecutar `applet` solamente; ignora todas las demás marcas de HTML. Si los sitios Web en el programa incluyeran `applet` de Java, sólo aparecerían esos `applet` en el `appletviewer` cuando el usuario seleccionara un sitio Web. Cada `applet` se ejecutaría en una ventana separada del `appletviewer`.

18.3 Cómo leer un archivo en un servidor Web

En nuestro siguiente ejemplo, una vez más ocultamos los detalles relacionados con la red. La aplicación de la figura 18.3 utiliza el componente de la GUI de Swing `JEditorPane` (del paquete `javax.swing`) para mostrar el contenido de un archivo en un servidor Web. El usuario introduce un URL en el objeto `JTextField` que se encuentra en la parte superior de la ventana, y el programa muestra el documento correspondiente (si es que existe) en el objeto `JEditorPane`. La clase `JEditorPane` puede desplegar tanto texto simple como texto con formato HTML, por lo que esta aplicación actúa como un navegador Web simple. La aplicación también demuestra cómo procesar eventos `HyperlinkEvent` cuando el usuario hace clic en un hipervínculo en el documento HTML. Las capturas de pantalla de la figura 18.3 muestran que el objeto `JEditorPane` puede mostrar tanto texto simple (la primera pantalla) como texto en HTML (la segunda pantalla). Las técnicas que se muestran en este ejemplo también pueden usarse en applet. Sin embargo, a los applet sólo se les permite leer archivos en el servidor del cual se hayan descargado.

La clase de aplicación `LeerArchivoServidor` contiene el objeto `JTextField` llamado `campoIntroducir`, en el cual el usuario escribe el URL del archivo a leer, y el objeto `JEditorPane` llamado `areaContenido` para mostrar el contenido del archivo. Cuando el usuario oprime la tecla *Intro* en `campo-`

```
1  // Fig. 18.3: LeerArchivoServidor.java
2  // Uso de un objeto JEditorPane para mostrar el contenido de un archivo en un servidor Web.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.net.*;
6  import java.io.*;
7  import javax.swing.*;
8  import javax.swing.event.*;
9
10 public class LeerArchivoServidor extends JFrame {
11    private JTextField campoIntroducir;
```

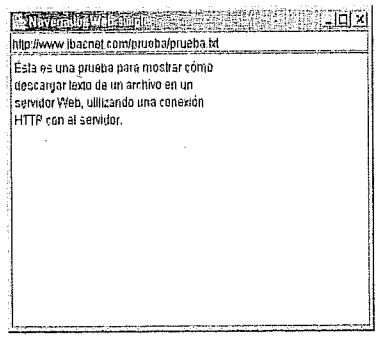
Figura 18.3 Cómo leer un archivo, abriendo una conexión a través de un `URL`. (Parte 1 de 3.)


```
12    private JEditorPane areaContenido;
13
14    // configurar la GUI
15    public LeerArchivoServidor()
16    {
17       super( "Navegador Web simple" );
18
19       Container contenedor = getContentPane();
20
21       // crear campoIntroducir y registrar su componente de escucha
22       campoIntroducir = new JTextField( "Escriba aquí el URL del archivo" );
23       campoIntroducir.addActionListener(
24          new ActionListener() {
25
26             // obtener el documento especificado por el usuario
27             public void actionPerformed( ActionEvent evento )
28             {
29                obtenerLaPagina( evento.getActionCommand() );
30             }
31
32          } // fin de la clase interna
33
34       ); // fin de la llamada a addActionListener
35
36       contenedor.add( campoIntroducir, BorderLayout.NORTH );
37
38       // crear areaContenido y registrar componente de escucha de evento HyperlinkEvent
39       areaContenido = new JEditorPane();
40       areaContenido.setEditable( false );
41       areaContenido.addHyperlinkListener(
42          new HyperlinkListener() {
43
44             // si el usuario hizo clic en el hipervínculo, ir a la página especificada
45             public void hyperlinkUpdate( HyperlinkEvent evento )
46             {
47                if ( evento.getEventType() ==
48                   HyperlinkEvent.EventType.ACTIVATED )
49                   obtenerLaPagina( evento.getURL().toString() );
50             }
51
52          } // fin de la clase interna
53
54       ); // fin de la llamada a addHyperlinkListener
55
56       contenedor.add( new JScrollPane( areaContenido ),
57          BorderLayout.CENTER );
58       setSize( 400, 300 );
59       setVisible( true );
60
61    } // fin del constructor de LeerArchivoServidor
62
63    // cargar documento
64    private void obtenerLaPagina( String ubicacion )
65    {
66       // cargar documento y mostrar ubicación
```

http://librosysolucionarios.net

Figura 18.3 Cómo leer un archivo, abriendo una conexión a través de un `URL`. (Parte 2 de 3.)

```
67          try {
68             areaContenido.setPage( ubicacion );
69             campoIntroducir.setText( ubicacion );
70          }
71          catch ( IOException excepcionES ) {
72             JOptionPane.showMessageDialog( this,
73                "Error al recuperar el URL especificado", "URL incorrecto",
74                JOptionPane.ERROR_MESSAGE );
75          }
76
77       } // fin del método obtenerLaPagina
78
79       public static void main( String args[] )
80       {
81          LeerArchivoServidor aplicacion = new LeerArchivoServidor();
82          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
83       }
84
85    } // fin de la clase LeerArchivoServidor
```



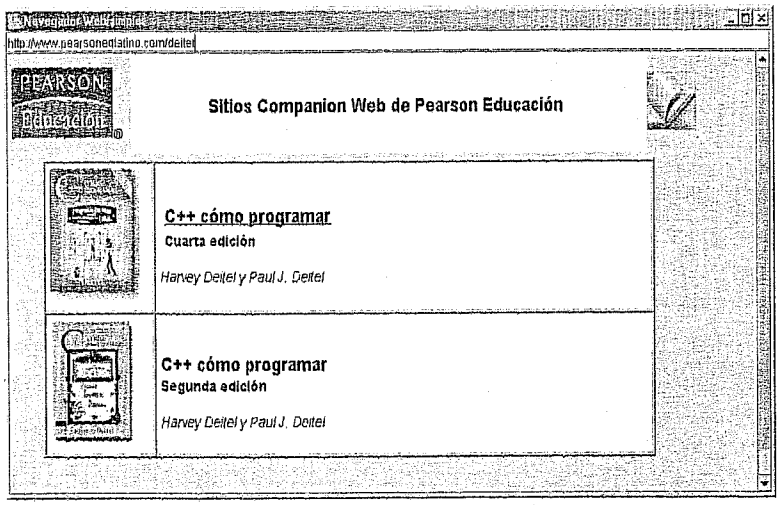


Figura 18.3 Cómo leer un archivo, abriendo una conexión a través de un `URL`. (Parte 3 de 3.)

http://librosysolucionarios.net

`Introducir`, el programa llama al método `actionPerformed` (líneas 27 a 30). En la línea 29 se utiliza el método `getActionCommand` de `ActionEvent` para obtener la cadena que introdujo el usuario en el objeto `JTextField`, y pasa esa cadena al método utilitario `obtenerLaPagina` (líneas 64 a 77).

En la línea 68 se utiliza el método ***`setPage`*** de `JEditorPane` para descargar el documento especificado por `ubicacion` y mostrarlo en el objeto `JEditorPane`. Si ocurre un error al descargar el documento, el método `setPage` lanza una excepción `IOException`. Además, si se especifica un URL inválido ocurre una excepción `MalformedURLException` (una subclase de `IOException`). Si el documento se carga correctamente, en la línea 69 se muestra la ubicación actual en `campoIntroducir`.

Por lo general, un documento HTML contiene *hipervínculos* (texto, imágenes o componentes de la GUI) que, cuando se hace clic sobre ellos, proporcionan un acceso rápido a otro documento en Web. Si un objeto `JEditorPane` contiene un documento HTML y el usuario hace clic en un hipervínculo, el objeto `JEditorPane` genera un evento ***`HyperlinkEvent`*** (paquete `javax.swing.event`) y notifica a todos los objetos ***`HyperlinkListener`*** (paquete `javax.swing.event`) registrados acerca de ese evento. En las líneas 41 a 54 se registra un objeto `HyperlinkListener` para manejar eventos `HyperlinkEvent`. Al ocurrir un evento `HyperlinkEvent`, el programa llama al método `hyperlinkUpdate` (líneas 45 a 50). En las líneas 47 y 48 se utiliza el método ***`getEventType`*** de `HyperlinkEvent` para determinar el tipo del evento `HyperlinkEvent`. La clase `HyperlinkEvent` contiene una clase anidada `public` llamada ***`EventType`***, la cual declara tres objetos `EventType` estáticos que representan los tipos de eventos de hipervínculos. ***`ACTIVATED`*** indica que el usuario hizo clic en un hipervínculo para cambiar de página Web, ***`ENTERED`*** indica que el usuario movió el ratón sobre un hipervínculo y ***`EXITED`*** indica que el usuario alejó el ratón de un hipervínculo. Si un hipervínculo fue activado (`ACTIVATED`), en la línea 49 se utiliza el método ***`getURL`*** de `HyperlinkEvent` para obtener el `URL` representado por el hipervínculo. El método `toString` convierte el `URL` devuelto en una cadena que puede pasarse al método utilitario `obtenerLaPagina`.

Observación de ingeniería de software 18.1

Un objeto `JEditorPane` genera eventos `HyperlinkEvent` solamente si no puede editarse.

18.4 Cómo establecer un servidor simple utilizando sockets de flujo

Los dos ejemplos descritos hasta ahora utilizan herramientas de red de alto nivel en Java para la comunicación entre las aplicaciones. En esos ejemplos no es responsabilidad del programador de Java establecer la conexión entre un cliente y un servidor. El primer programa dependió del navegador Web para comunicarse con un servidor Web. El segundo dependió de un objeto `JEditorPane` para realizar la conexión. En esta sección comenzaremos con nuestra discusión acerca de cómo crear sus propias aplicaciones que puedan comunicarse entre sí.

Para establecer un servidor simple en Java se requieren cinco pasos. El primero es crear un objeto ***`ServerSocket`***. Una llamada al constructor de `ServerSocket` como:

```
ServerSocket servidor = new ServerSocket( puerto, longitudCola );
```

registra un *número de puerto* disponible y especifica el máximo número de clientes que pueden esperar para conectarse al servidor (es decir, la *longitudCola*). El número de puerto es utilizado por los clientes para localizar la aplicación servidor en el equipo servidor. A menudo, a esto se le conoce como *punto de negociación (handshake)*. Si la cola está llena, el servidor rechaza las conexiones de los clientes. El constructor establece el puerto en donde el servidor espera las conexiones de los clientes; a este proceso se le conoce como *enlazar el servidor al puerto*. Cada cliente pedirá conectarse con el servidor en este *puerto*. Sólo una aplicación puede enlazarse a un puerto específico en el servidor, en un momento dado.

Observación de ingeniería de software 18.2

Los números de puerto pueden ser entre 0 y 65,535. Algunos sistemas operativos reservan los números de puertos menores que 1024 para los servicios del sistema (como los servidores de e-mail y World Wide Web). Por lo general, estos puertos no deben especificarse como puertos de conexión en los programas de los usuarios. De hecho, algunos sistemas operativos requieren de privilegios de acceso especiales para enlazarse a los números de puerto menores que 1024.

Los programas administran cada conexión cliente mediante un objeto `Socket`. En el paso 2, el servidor escucha indefinidamente (o *bloquea*) para esperar a que un cliente trate de conectarse. Para escuchar una conexión de un cliente, el programa llama al método `accept` de `ServerSocket`, como se muestra a continuación:

```
Socket conexion = servidor.accept();
```

esta instrucción devuelve un objeto `Socket` cuando se establece la conexión con un cliente. El socket permite al servidor interactuar con el cliente. El puerto especificado en el paso 1 puede utilizarse nuevamente en un servidor con subprocesamiento múltiple, para aceptar otra conexión cliente. En la sección 18.8 demostraremos este concepto.

El paso 3 es obtener los objetos `OutputStream` e `InputStream` que permiten al servidor comunicarse con el cliente, enviando y recibiendo bytes. El servidor envía información al cliente mediante un objeto `OutputStream`. El servidor recibe información del cliente mediante un objeto `InputStream`. El servidor invoca al método `getOutputStream` en el objeto `Socket` para obtener una referencia al objeto `OutputStream` del objeto `Socket`, e invoca al método `getInputStream` en el objeto `Socket` para obtener una referencia al objeto `InputStream` del objeto `Socket`.

Los objetos flujo pueden utilizarse para enviar o recibir bytes individuales, o conjuntos de bytes, mediante el método `write` de `OutputStream` y el método `read` de `InputStream`, respectivamente. A menudo es útil enviar o recibir valores de tipos primitivos (como `int` y `double`) u objetos `Serializable` (como objetos `String` u otros tipos serializables), en vez de enviar bytes. En este caso podemos utilizar las técnicas del capítulo 17 para envolver otros tipos de flujos (como `ObjectOutputStream` y `ObjectInputStream`) alrededor de los objetos `OutputStream` e `InputStream` asociados con el objeto `Socket`. Por ejemplo,

```
ObjectInputStream entrada =
   new ObjectInputStream( conexion.getInputStream() );

ObjectOutputStream salida =
   new ObjectOutputStream( conexion.getOutputStream() );
```

Lo mejor de establecer estas relaciones es que, cualquier cosa que escriba el servidor en el objeto `ObjectOutputStream` se enviará mediante el objeto `OutputStream` y estará disponible en el objeto `InputStream` del cliente, y cualquier cosa que el cliente escriba en su objeto `OutputStream` (mediante su correspondiente objeto `ObjectOutputStream`) estará disponible a través del objeto `InputStream` del servidor. La transmisión de los datos a través de la red es un proceso transparente, y se maneja completamente mediante Java.

El paso 4 es la fase de *procesamiento*, en la cual el servidor y el cliente se comunican a través de los objetos `OutputStream` e `InputStream`. En el paso 5, cuando se completa la transmisión, el servidor cierra la conexión invocando al método `close` en los flujos y en el objeto `Socket`.



Observación de ingeniería de software 18.3

Con los sockets, la E/S de red es vista por los programas de Java como algo similar a un archivo de E/S secuencial. Los sockets ocultan al programador gran parte de la complejidad de la programación en red.



Observación de ingeniería de software 18.4

Mediante el subprocesamiento múltiple en Java, podemos crear servidores que utilicen esta característica y puedan administrar muchas conexiones simultáneas con muchos clientes; esta arquitectura de servidor con subprocesamiento múltiple es precisamente lo que utilizan los servidores de red populares.



Observación de ingeniería de software 18.5

Un servidor con subprocesamiento múltiple puede tomar el `Socket` devuelto por cada llamada al método `accept`, y puede crear un nuevo subproceso que administre la E/S de red a través de ese objeto `Socket`, o un servidor con subprocesamiento múltiple puede mantener una reserva de subprocesos (un conjunto de subprocesos ya existentes) listos para administrar la E/S de red a través de los nuevos objetos `Socket`, a medida que se vayan creando.



Tip de rendimiento 18.2

En los sistemas de alto rendimiento en los que la memoria es abundante, puede implementar un servidor con subprocesamiento múltiple para crear una reserva de subprocesos que puedan asignarse rápidamente para manejar la

http://librosysolucionarios.net

E/S de red a través de cada nuevo `Socket`, a medida que se vayan creando. Por lo tanto, cuando el servidor recibe una conexión no necesita incurrir en la sobrecarga que se genera debido a la creación de los subprocesos. Cuando se cierra la conexión, el subproceso se devuelve a la reserva para su reutilización.

18.5 Cómo establecer un cliente simple utilizando sockets de flujo

Para establecer un cliente simple en Java se requieren cuatro pasos. En el primero creamos un objeto `Socket` para conectarse al servidor. El constructor de `Socket` establece la conexión al servidor. Por ejemplo, la instrucción:

```
Socket conexion = new Socket( direccionServidor, puerto );
```

utiliza el constructor de `Socket` con dos argumentos: la dirección IP del servidor (*direccionServidor*) y el número de *puerto*. Si el intento de conexión es exitoso, esta instrucción devuelve un objeto `Socket`. Un intento de conexión fallido lanzará una instancia de una subclase de `IOException`, por lo que muchos programas simplemente atrapan a `IOException`. Una excepción `UnknownHostException` ocurre cuando el sistema no puede resolver la dirección del servidor especificada en la llamada al constructor de `Socket`. Una excepción `ConnectException` se lanza cuando ocurre un error mientras se trata de hacer una conexión a un servidor.

En el paso 2, el cliente utiliza los métodos `getInputStream` y `getOutputStream` de la clase `Socket` para obtener referencias a los objetos `InputStream` y `OutputStream` de `Socket`. En este caso podemos utilizar las técnicas del capítulo 17 para envolver otros tipos de flujos (como `ObjectOutputStream` y `ObjectInputStream`) alrededor de los objetos `OutputStream` e `InputStream` asociados con el objeto `Socket`. Si el servidor va a enviar información en el formato de los tipos actuales, el cliente debe recibir esa información en el mismo formato. Por lo tanto, si el servidor envía los valores con un objeto `ObjectOutputStream`, el cliente debe leer esos valores con un objeto `ObjectInputStream`.

El paso 3 es la fase de procesamiento, en la cual el cliente y el servidor se comunican a través de los objetos `InputStream` y `OutputStream`. En el paso 4, el cliente cierra la conexión cuando se completa la transmisión, invocando al método `close` en los flujos y en el objeto `Socket`. Al procesar la información enviada por un servidor, el cliente debe determinar cuándo va a terminar el servidor de enviar información, de manera que pueda llamar al método `close` para cerrar la conexión del objeto `Socket`. Por ejemplo, el método `read` de `InputStream` devuelve el valor –1 cuando detecta el fin del flujo (lo que se conoce también como EOF: fin del archivo). Si se utiliza un objeto `ObjectInputStream` para leer información del servidor, se produce una excepción `EOFException` cuando el cliente trata de leer un valor de un flujo en el que se detectó el fin del flujo.

18.6 Interacción entre cliente/servidor mediante conexiones de socket de flujo

En la figura 18.4 y 18.5 utilizamos *sockets de flujo* para demostrar una *aplicación cliente/servidor para conversar*. El servidor espera un intento de conexión por parte de un cliente. Cuando se conecta un cliente al servidor, la aplicación servidor envía un objeto `String` (recuerde que los objetos `String` son objetos `Serializable`) indicando que la conexión con el cliente fue exitosa. Después el cliente muestra el mensaje. Ambas aplicaciones cliente y servidor proporcionan campos de texto que permiten al usuario escribir un mensaje y enviarlo de una a otra aplicación. Cuando el cliente o el servidor envían la cadena `"TERMINAR"`, la conexión entre el cliente y el servidor termina. Después el servidor espera a que se conecte otro cliente. La declaración de la clase `Servidor` aparece en la figura 18.4. La declaración de la clase `Cliente` aparece en la figura 18.5. Las capturas de pantalla en las que se muestra la ejecución entre el cliente y el servidor aparecen como parte de la figura 18.5.

La clase `Servidor`

El constructor de `Servidor` (líneas 20 a 51) crea la GUI del servidor, la cual contiene un objeto `JTextField` y un objeto `JTextArea`. `Servidor` muestra su salida en el objeto `JTextArea`. Cuando se ejecuta el método `main` (líneas 210 a 215) crea un objeto `Servidor`, especifica la operación de cierre predeterminada de la ventana y llama al método `ejecutarServidor` (declarado en las líneas 54 a 89).

```
1  // Fig. 18.4: Servidor.java
2  // Configurar un servidor que reciba una conexión de un cliente, envíe
3  // una cadena al cliente y cierre la conexión.
4  import java.io.*;
5  import java.net.*;
6  import java.awt.*;
7  import java.awt.event.*;
8  import javax.swing.*;
9
10 public class Servidor extends JFrame {
11    private JTextField campoIntroducir;
12    private JTextArea areaPantalla;
13    private ObjectOutputStream salida;
14    private ObjectInputStream entrada;
15    private ServerSocket servidor;
16    private Socket conexion;
17    private int contador = 1;
18
19    // configurar la GUI
20    public Servidor()
21    {
22       super( "Servidor" );
23
24       Container contenedor = getContentPane();
25
26       // crear campoIntroducir y registrar componente de escucha
27       campoIntroducir = new JTextField();
28       campoIntroducir.setEditable( false );
29       campoIntroducir.addActionListener(
30          new ActionListener() {
31
32             // enviar mensaje al cliente
33             public void actionPerformed( ActionEvent evento )
34             {
35                enviarDatos( evento.getActionCommand() );
36                campoIntroducir.setText( "" );
37             }
38          }
39       );
40
41       contenedor.add( campoIntroducir, BorderLayout.NORTH );
42
43       // crear areaPantalla
44       areaPantalla = new JTextArea();
45       contenedor.add( new JScrollPane( areaPantalla ),
46          BorderLayout.CENTER );
47
48       setSize( 300, 150 );
49       setVisible( true );
50
51    } // fin del constructor de Servidor
52
53    // configurar y ejecutar el servidor
54    public void ejecutarServidor()
```

Figura 18.4 Porción correspondiente al servidor de una conexión cliente/servidor basada en sockets. (Parte 1 de 4.)

http://libros ysoluciones.net


```
55    {
56       // configurar servidor para que reciba conexiones; procesar las conexiones
57       try {
58
59          // Paso 1: crear un objeto ServerSocket.
60          servidor = new ServerSocket( 12345, 100 );
61
62          while ( true ) {
63
64             try {
65                esperarConexion();   // Paso 2: esperar una conexión
66                obtenerFlujos();     // Paso 3: obtener flujos de entrada y salida
67                procesarConexion(); // Paso 4: procesar la conexión
68             }
69
70             // procesar excepción EOFException cuando el cliente cierre la conexión
71             catch ( EOFException excepcionEOF ) {
72                System.err.println( "El servidor terminó la conexión" );
73             }
74
75             finally {
76                cerrarConexion();   // Paso 5: cerrar la conexión.
77                ++contador;
78             }
79
80          } // fin de instrucción while
81
82       } // fin del bloque try
83
84       // procesar problemas con E/S
85       catch ( IOException excepcionES ) {
86          excepcionES.printStackTrace();
87       }
88
89    } // fin del método ejecutarServidor
90
91    // esperar que la conexión llegue, después mostrar información de la conexión
92    private void esperarConexion() throws IOException
93    {
94       mostrarMensaje( "Esperando una conexión\n" );
95       conexion = servidor.accept(); // permitir al servidor aceptar la conexión
96       mostrarMensaje( "Conexión " + contador + " recibida de: " +
97          conexion.getInetAddress().getHostName() );
98    }
99
100   // obtener flujos para enviar y recibir datos
101   private void obtenerFlujos() throws IOException
102   {
103      // establecer flujo de salida para los objetos
104      salida = new ObjectOutputStream( conexion.getOutputStream() );
105      salida.flush(); // vaciar búfer de salida para enviar información de encabezado
106
107      // establecer flujo de entrada para los objetos
108      entrada = new ObjectInputStream( conexion.getInputStream() );
```

Figura 18.4 Porción correspondiente al servidor de una conexión cliente/servidor basada en sockets. (Parte 2 de 4.)

```
109
110         mostrarMensaje( "\nSe recibieron los flujos de E/S\n" );
111      }
112
113      // procesar la conexión con el cliente
114      private void procesarConexion() throws IOException
115      {
116         // enviar mensaje de conexión exitosa al cliente
117         String mensaje = "Conexión exitosa";
118         enviarDatos( mensaje );
119
120         // habilitar campoIntroducir para que el usuario del servidor pueda enviar mensajes
121         establecerCampoTextoEditable( true );
122
123         do { // procesar los mensajes enviados por el cliente
124
125            // leer el mensaje y mostrarlo en pantalla
126            try {
127               mensaje = ( String ) entrada.readObject();
128               mostrarMensaje( "\n" + mensaje );
129            }
130
131            // atrapar problemas que pueden ocurrir al tratar de leer del cliente
132            catch ( ClassNotFoundException excepcionClaseNoEncontrada ) {
133               mostrarMensaje( "\nSe recibió un tipo de objeto desconocido" );
134            }
135
136         } while ( !mensaje.equals( "CLIENTE>>> TERMINAR" ) );
137
138      } // fin del método procesarConexion
139
140      // cerrar flujos y socket
141      private void cerrarConexion()
142      {
143         mostrarMensaje( "\nFinalizando la conexión\n" );
144         establecerCampoTextoEditable( false ); // deshabilitar campoIntroducir
145
146         try {
147            salida.close();
148            entrada.close();
149            conexion.close();
150         }
151         catch( IOException excepcionES ) {
152            excepcionES.printStackTrace();
153         }
154      }
155
156      // enviar mensaje al cliente
157      private void enviarDatos( String mensaje )
158      {
159         // enviar objeto al cliente
160         try {
161            salida.writeObject( "SERVIDOR>>> " + mensaje );
162            salida.flush();
```

Figura 18.4 Porción correspondiente al servidor de una conexión cliente/servidor basada en sockets. (Parte 3 de 4.)

http://librosysolucionarios.net


```
163            mostrarMensaje( "\nSERVIDOR>>> " + mensaje );
164         }
165
166         // procesar problemas que pueden ocurrir al enviar el objeto
167         catch ( IOException excepcionES ) {
168            areaPantalla.append( "\nError al escribir objeto" );
169         }
170      }
171
172      // método utilitario que es llamado desde otros subprocesos para manipular a
173      // areaPantalla en el subproceso despachador de eventos
174      private void mostrarMensaje( final String mensajeAMostrar )
175      {
176         // mostrar mensaje del subproceso de ejecución despachador de eventos
177         SwingUtilities.invokeLater(
178            new Runnable() {      // clase interna para asegurar que la GUI
179            se actualice apropiadamente
180               public void run() // actualiza areaPantalla
181               {
182                  areaPantalla.append( mensajeAMostrar );
183                  areaPantalla.setCaretPosition(
184                     areaPantalla.getText().length() );
185               }
186
187            } // fin de la clase interna
188
189         ); // fin de la llamada a SwingUtilities.invokeLater
190      }
191
192      // método utilitario que es llamado desde otros subprocesos para manipular a
193      // campoIntroducir en el subproceso despachador de eventos
194      private void establecerCampoTextoEditable( final boolean editable )
195      {
196         // mostrar mensaje del subproceso de ejecución despachador de eventos
197         SwingUtilities.invokeLater(
198            new Runnable() {  // clase interna para asegurar que la GUI
199            se actualice apropiadamente
200               public void run() // establece la capacidad de modificar a campoIntroducir
201               {
202                  campoIntroducir.setEditable( editable );
203               }
204
205            } // fin de la clase interna
206
207         ); // fin de la llamada a SwingUtilities.invokeLater
208      }
209
210      public static void main( String args[] )
211      {
212         Servidor aplicacion = new Servidor();
213         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
214         aplicacion.ejecutarServidor();
215      }
216
217   } // fin de la clase Servidor
```

Figura 18.4 Porción correspondiente al servidor de una conexión cliente/servidor basada en sockets. (Parte 4 de 4.)

El método `ejecutarServidor` configura el servidor para que reciba una conexión y procese una conexión a la vez. En la línea 60 se crea un objeto `ServerSocket` llamado `servidor`, para que espere las conexiones. El objeto `ServerSocket` escucha en el puerto `12345`, en espera de que un cliente se conecte. El segundo argumento para el constructor es el número de conexiones que pueden esperar en una cola para conectarse al servidor (`100` en este ejemplo). Si la cola está llena cuando un cliente trate de conectarse, el servidor rechazará la conexión.

Error común de programación 18.1

Al especificar un puerto que ya esté en uso, o especificar un número de puerto incorrecto al crear un objeto `ServerSocket`, se produce una excepción `IllegalArgumentException`

En la línea 65 se hace una llamada al método `esperarConexion` (declarado en las líneas 92 a 98) para esperar la conexión de un cliente. Una vez establecida la conexión, en la línea 66 se hace una llamada al método `obtenerFlujos` (declarado en las líneas 101 a 111) para obtener las referencias a los flujos para la conexión. En la línea 67 se hace una llamada al método `procesarConexion` (declarado en las líneas 114 a 138) para enviar el mensaje de conexión inicial al cliente, y para procesar todos los mensajes que se reciban del cliente. El bloque `finally` (líneas 75 a 78) finaliza la conexión del cliente llamando al método `cerrarConexion` (líneas 141 a 154), incluso aunque haya ocurrido una excepción. El método `mostrarMensaje` (líneas 174 a 190) es llamado desde estos métodos para utilizar el subproceso despachador de eventos para mostrar los mensajes en el área de texto de la aplicación. En las líneas 183 y 194 del método `areaPantalla` se utiliza el método `setCaretPosition` de `JTextComponent` para posicionar el cursor de entrada en el área de texto, después del último carácter mostrado en esa área de texto. Al hacer esto, el área de texto se desplaza a medida que se le va anexando texto.

En el método `esperarConexion` (líneas 92 a 98), en la línea 95 se utiliza el método `accept` de `ServerSocket` para esperar una conexión de un cliente. Al ocurrir una conexión, el objeto `Socket` resultante se asigna a `conexion`. El método `accept` realiza un bloqueo hasta que se reciba una conexión (es decir, el subproceso en el que se haga la llamada a `accept` detiene su ejecución hasta que un cliente se conecte). En las líneas 96 y 97 se imprime en pantalla el nombre del equipo que realizó la conexión. El método `getInetAddress` de `Socket` devuelve un objeto `InetAddress` (paquete `java.net`), el cual contiene información acerca del equipo cliente. El método `getHostName` de `InetAddress` devuelve el nombre de host del equipo cliente. Por ejemplo, hay una dirección IP (`127.0.0.1`) y nombre de host (`localhost`) especiales, que son útiles para probar aplicaciones de red en su equipo local. Si se hace una llamada a `getHostName` en un objeto `InetAddress` que contenga `127.0.0.1`, el nombre de host correspondiente que devuelve el método es `localhost`.

El método `obtenerFlujos` (líneas 101 a 111) obtiene las referencias a los flujos de `Socket` y los utiliza para inicializar un objeto `ObjectOutputStream` (línea 104) y un objeto `ObjectInputStream` (línea 108), respectivamente. Observe la llamada al método `flush` de `ObjectOutputStream` en la línea 105. Esta instrucción hace que el objeto `ObjectOutputStream` en el servidor envíe un *encabezado de flujo* al objeto `ObjectInputStream` correspondiente del cliente. El encabezado de flujo contiene información tal como la versión de la serialización de objetos que se va a utilizar para enviar los objetos. Esta información es requerida por el objeto `ObjectInputStream`, para que pueda prepararse para recibir estos objetos de manera correcta.

Observación de ingeniería de software 18.6

Al utilizar un objeto `ObjectOutputStream` y un objeto `ObjectInputStream` para enviar y recibir datos a través de una conexión de red, siempre debe crear primero el objeto `ObjectOutputStream` y vaciar (mediante el método `flush`) el flujo, de manera que el objeto `ObjectInputStream` pueda prepararse para recibir los datos. Esto se requiere sólo en las aplicaciones de red que se comunican utilizando a `ObjectOutputStream` y `ObjectInputStream`.

Tip de rendimiento 18.3

Los búferes de salida se utilizan comúnmente para incrementar la eficiencia de una aplicación, al enviar mayores cantidades de datos con menos frecuencia. Por lo general, los componentes de entrada y salida de hardware son más lentos que la memoria de la computadora.

http://librosysolucionarios.net

En la línea 118 del método `procesarConexion` (líneas 114 a 138) se hace una llamada al método `enviarDatos` para enviar la cadena "`SERVIDOR>>> Conexión exitosa`" al cliente. El ciclo de las líneas 123 a 136 se ejecuta hasta que el servidor recibe el mensaje "`CLIENTE>>> TERMINAR`." En la línea 127 se utiliza el método `readObject` de `ObjectInputStream` para leer un objeto `String` del cliente. En la línea 128 se invoca el método `mostrarMensaje` para anexar el mensaje al objeto `JTextArea`.

Cuando termina la transmisión, el método `procesarConexion` regresa y el programa llama al método `cerrarConexion` (líneas 141 a 154) para cerrar los flujos asociados con el objeto `Socket`, y para cerrar también a ese objeto `Socket`. Después, el servidor espera el siguiente intento de conexión por parte de un cliente, continuando con la línea 65 al principio del ciclo `while`.

Cuando el usuario de la aplicación servidor introduce una cadena en el campo de texto y oprime la tecla *Intro*, el programa llama al método `actionPerformed` (líneas 33 a 37), el cual lee la cadena del campo de texto y llama al método utilitario `enviarDatos` (líneas 157 a 170) para enviar la cadena al cliente. El método `enviarDatos` escribe el objeto, vacía el búfer de salida y anexa la misma cadena al área de texto en la ventana del servidor. No es necesario invocar a `mostrarMensaje` para modificar el área de texto aquí, ya que el método `enviarDatos` es llamado desde un manejador de eventos; por lo tanto, `enviarDatos` se ejecuta como parte del subproceso despachador de eventos.

Observe que el objeto `Servidor` recibe una conexión, la procesa, cierra la conexión y espera a la siguiente conexión. Un escenario más apropiado sería un objeto `Servidor` que recibe una conexión, la configura para procesarla como un subproceso de ejecución separado, e inmediatamente se pone en espera de nuevas conexiones. Los subprocesos separados que procesan conexiones existentes pueden seguir ejecutándose, mientras que el `Servidor` se concentra en nuevas peticiones de conexión. Esto hace al servidor más eficiente, ya que pueden procesarse varias peticiones de clientes en forma concurrente. En la sección 18.8 mostraremos el uso de un servidor con subprocesamiento múltiple.

La clase `Cliente`

Al igual que la clase `Servidor`, el constructor de la clase `Cliente` (figura 18.5) crea la GUI de la aplicación (un objeto `JTextField` y un objeto `JTextArea`). `Cliente` muestra su salida en el área de texto. Cuando se ejecuta el método `main` (líneas 199 a 210), se crea una instancia de la clase `Cliente`, se especifica la operación de cierre predeterminada de la ventana y se hace una llamada al método `ejecutarCliente` (declarado en las líneas 55 a 78). En este ejemplo, usted puede ejecutar el cliente desde cualquier computadora en Internet, y especificar la dirección IP o nombre de host del equipo servidor como un argumento de línea de comandos para el programa. Por ejemplo, el comando:

```
java Cliente 192.168.1.15
```

intenta realizar una conexión al objeto `Servidor` en la computadora que tiene la dirección IP `192.168.1.15`.

```
1  // Fig. 18.5: Cliente.java
2  // Cliente que lee y muestra la información que le envía un Servidor.
3  import java.io.*;
4  import java.net.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class Cliente extends JFrame {
10    private JTextField campoIntroducir;
11    private JTextArea areaPantalla;
12    private ObjectOutputStream salida;
```

Figura 18.5 Porción correspondiente al cliente de una conexión basada en sockets entre un cliente y un servidor. (Parte 1 de 6.)

```
13    private ObjectInputStream entrada;
14    private String mensaje = "";
15    private String servidorChat;
16    private Socket cliente;
17
18    // inicializar servidorChat y configurar la GUI
19    public Cliente( String host )
20    {
21       super( "Cliente" );
22
23       servidorChat = host; // establecer el servidor al que se va a conectar este cliente
24
25       Container contenedor = getContentPane();
26
27       // crear campoIntroducir y registrar componente de escucha
28       campoIntroducir = new JTextField();
29       campoIntroducir.setEditable( false );
30       campoIntroducir.addActionListener(
31          new ActionListener() {
32
33             // enviar mensaje al servidor
34             public void actionPerformed( ActionEvent evento )
35             {
36                enviarDatos( evento.getActionCommand() );
37                campoIntroducir.setText( "" );
38             }
39          }
40       );
41
42       contenedor.add( campoIntroducir, BorderLayout.NORTH );
43
44       // crear areaPantalla
45       areaPantalla = new JTextArea();
46       contenedor.add( new JScrollPane( areaPantalla ),
47          BorderLayout.CENTER );
48
49       setSize( 300, 150 );
50       setVisible( true );
51
52    } // fin del constructor de Cliente
53
54    // conectarse al servidor y procesar mensajes del servidor
55    private void ejecutarCliente()
56    {
57       // conectarse al servidor, obtener flujos, procesar la conexión
58       try {
59          conectarAServidor(); // Paso 1: crear un socket para realizar la conexión
60          obtenerFlujos();     // Paso 2: obtener los flujos de entrada y salida
61          procesarConexion();  // Paso 3: procesar la conexión
62       }
63
64       // el servidor cerró la conexión
65       catch ( EOFException excepcionEOF ) {
66          System.err.println( "El cliente terminó la conexión" );
```

Figura 18.5 Porción correspondiente al cliente de una conexión basada en sockets entre un cliente y un servidor. (Parte 2 de 6.)


```
67       }
68
69       // procesar los problemas que pueden ocurrir al comunicarse con el servidor
70       catch ( IOException excepcionES ) {
71          excepcionES.printStackTrace();
72       }
73
74       finally {
75          cerrarConexion(); // Paso 4: cerrar la conexión
76       }
77
78    } // fin del método ejecutarCliente
79
80    // conectarse al servidor
81    private void conectarAServidor() throws IOException
82    {
83       mostrarMensaje( "Intentando realizar conexión\n" );
84
85       // crear Socket para realizar la conexión con el servidor
86       cliente = new Socket( InetAddress.getByName( servidorChat ), 12345 );
87
88       // mostrar la información de la conexión
89       mostrarMensaje( "Conectado a: " +
90          cliente.getInetAddress().getHostName() );
91    }
92
93    // obtener flujos para enviar y recibir datos
94    private void obtenerFlujos() throws IOException
95    {
96       // establecer flujo de salida para los objetos
97       salida = new ObjectOutputStream( cliente.getOutputStream() );
98       salida.flush(); // vaciar búfer de salida para enviar información de encabezado
99
100      // establecer flujo de entrada para los objetos
101      entrada = new ObjectInputStream( cliente.getInputStream() );
102
103      mostrarMensaje( "\nSe recibieron los flujos de E/S\n" );
104   }
105
106   // procesar la conexión con el servidor
107   private void procesarConexion() throws IOException
108   {
109      // habilitar campoIntroducir para que el usuario del cliente pueda enviar mensajes
110      establecerCampoTextoEditable( true );
111
112      do { // procesar mensajes enviados del servidor
113
114         // leer mensaje y mostrarlo en pantalla
115         try {
116            mensaje = ( String ) entrada.readObject();
117            mostrarMensaje( "\n" + mensaje );
118         }
119
120         // atrapar los problemas que pueden ocurrir al leer del servidor
```

Figura 18.5 Porción correspondiente al cliente de una conexión basada en sockets entre un cliente y un servidor. (Parte 3 de 6.)

http://librosysolucionarios.net

```
            catch ( ClassNotFoundException excepcionClaseNoEncontrada ) {
               mostrarMensaje( "\nSe recibió un objeto de tipo desconocido" );
            }

         } while ( !mensaje.equals( "SERVIDOR>>> TERMINAR" ) );

      } // fin del método procesarConexion

      // cerrar flujos y socket
      private void cerrarConexion()
      {
         mostrarMensaje( "\nCerrando conexión" );
         establecerCampoTextoEditable( false ); // deshabilitar campoIntroducir

         try {
            salida.close();
            entrada.close();
            cliente.close();
         }
         catch( IOException excepcionES ) {
            excepcionES.printStackTrace();
         }
      }

      // enviar mensaje al servidor
      private void enviarDatos( String mensaje )
      {
         // enviar objeto al servidor
         try {
            salida.writeObject( "CLIENTE>>> " + mensaje );
            salida.flush();
            mostrarMensaje( "\nCLIENTE>>> " + mensaje );
         }

         // procesar los problemas que pueden ocurrir al enviar el objeto
         catch ( IOException excepcionES ) {
            areaPantalla.append( "\nError al escribir el objeto" );
         }
      }

      // método utilitario que es llamado desde otros subprocesos para manipular a
      // areaPantalla en el subproceso despachador de eventos
      private void mostrarMensaje( final String mensajeAMostrar )
      {
         // mostrar mensaje del subproceso de ejecución de la GUI
         SwingUtilities.invokeLater(
            new Runnable() { // clase interna para asegurar que la GUI se actualice
            apropiadamente

               public void run() // actualiza areaPantalla
               {
                  areaPantalla.append( mensajeAMostrar );
                  areaPantalla.setCaretPosition(
                     areaPantalla.getText().length() );
               }
```

Figura 18.5 Porción correspondiente al cliente de una conexión basada en sockets entre un cliente y un servidor. (Parte 4 de 6.)

http://librosysolucionarios.net


```
            } // fin de la clase interna

         }; // fin de la llamada a SwingUtilities.invokeLater
      }

      // método utilitario que es llamado desde otros subprocesos para manipular a
      // campoIntroducir en el subproceso despachador de eventos
      private void establecerCampoTextoEditable( final boolean editable )
      {
         // mostrar mensaje del subproceso de ejecución de la GUI
         SwingUtilities.invokeLater(
            new Runnable() { // clase interna para asegurar que la GUI se actualice
            apropiadamente

               public void run() // establece la capacidad de modificar campoIntroducir
               {
                  campoIntroducir.setEditable( editable );
               }

            } // fin de la clase interna

         ); // fin de la llamada a SwingUtilities.invokeLater
      }

      public static void main( String args[] )
      {
         Cliente aplicacion;

         if ( args.length == 0 )
            aplicacion = new Cliente( "127.0.0.1" );
         else
            aplicacion = new Cliente( args[ 0 ] );

         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
         aplicacion.ejecutarCliente();
      }

   } // fin de la clase Cliente
```

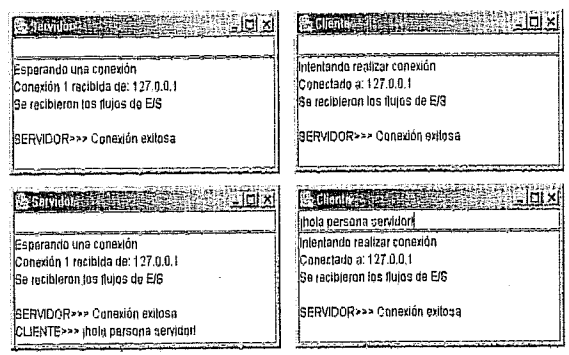


Figura 18.5 Porción correspondiente al cliente de una conexión basada en sockets entre un cliente y un servidor. (Parte 5 de 6.)

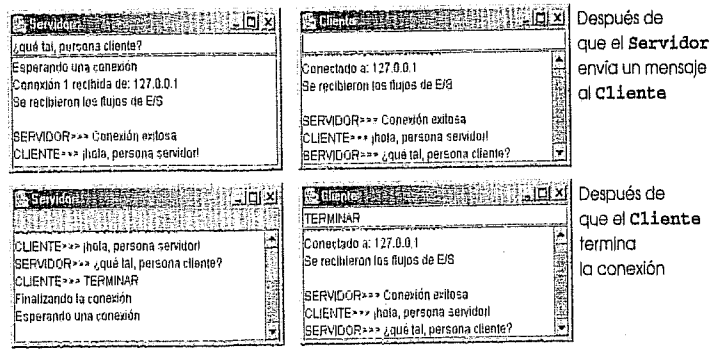


Figura 18.5 Porción correspondiente al cliente de una conexión basada en sockets entre un cliente y un servidor. (Parte 6 de 6.)

El método `ejecutarCliente` (líneas 55 a 78) de la clase `Cliente` establece la conexión con el servidor, procesa los mensajes recibidos del servidor y cierra la conexión cuando termina la comunicación. En la línea 59 se hace una llamada al método `conectarAServidor` (declarado en las líneas 81 a 91) para realizar la conexión. Después de realizar la conexión, en la línea 60 se hace una llamada al método `obtenerFlujos` (declarado en las líneas 94 a 104) para obtener las referencias a los objetos flujo del objeto `Socket`. Después, en la línea 61 se hace una llamada al método `procesarConexion` (declarado en las líneas 107 a 127) para recibir y mostrar los mensajes enviados por el servidor. En el bloque `finally` (líneas 74 a 76) se hace una llamada al método `cerrarConexion` (líneas 130 a 143) para cerrar los flujos y el objeto Socket, incluso aunque haya ocurrido una excepción. El método `mostrarMensaje` (líneas 163 a 179) es llamado desde estos métodos para utilizar el subproceso despachador de eventos para mostrar los mensajes en el área de texto de la aplicación.

El método `conectarAServidor` (líneas 81 a 91) crea un objeto `Socket` llamado `cliente` (línea 86) para establecer una conexión. Este método pasa dos argumentos al constructor de `Socket`: la dirección IP del equipo servidor y el número de puerto (12345) en donde la aplicación servidor está esperando las conexiones de los clientes. En el primer argumento, el método estático `getByName` de `InetAddress` devuelve un objeto `InetAddress` que contiene la dirección IP especificada como argumento en la línea de comandos para la aplicación (o `127.0.0.1` si no se especifican argumentos en la línea de comandos). El método `getByName` puede recuperar una cadena que contiene la dirección IP actual, o el nombre de host del servidor. El primer argumento también podría haberse escrito de otras formas. Para la dirección 127.0.0.1 de localhost, el primer argumento podría ser:

```
InetAddress.getByName( "localhost" )
```

o

```
InetAddress.getLocalHost()
```

Además, hay versiones del constructor de `Socket` que reciben una cadena para la dirección IP o nombre de host. El primer argumento podía haberse especificado como `"127.0.0.1"` o `"localhost"`.[1] El segundo argumento del constructor de `Socket` es el número de puerto del servidor. Este número debe concordar con el número de puerto en el que el servidor esté esperando las conexiones, al cual se le conoce como el punto de negociación (handshake). Una vez que se realiza la conexión, en las líneas 89 y 90 se muestra un mensaje en el área de texto, indicando el nombre del equipo servidor al cual se conectó el cliente.

1. Optamos por demostrar la relación cliente/servidor conectándonos entre programas ejecutándose en el mismo equipo (localhost). Por lo general, este primer argumento es la dirección IP de otra computadora. El objeto `InetAddress` de otra computadora puede obtenerse especificando la dirección IP o nombre de host del otro equipo, como argumento para el método `getByName` de `InetAddress`.

El objeto `Cliente` utiliza un objeto `ObjectOutputStream` para enviar datos al servidor, y un objeto `ObjectInputStream` para recibir datos del servidor. El método `obtenerFlujos` (líneas 94 a 104) crea los objetos `ObjectOutputStream` y `ObjectInputStream` que utilizan los flujos asociados con el socket de `cliente`.

El método `procesarConexion` (líneas 107 a 127) contiene un ciclo que se ejecuta hasta que el cliente recibe el mensaje `"SERVIDOR>>> TERMINAR"`. En la línea 116 se lee un objeto `String` del servidor. En la línea 117 se invoca el método `mostrarMensaje` para anexar el mensaje al área de texto.

Cuando la transmisión termina, el método `cerrarConexion` (líneas 130 a 143) cierra los flujos y el objeto `Socket`.

Cuando el usuario de la aplicación cliente introduce una cadena en el campo de texto y oprime la tecla *Intro*, el programa llama al método `actionPerformed` (líneas 34 a 38) para leer la cadena del campo de texto e invoca al método utilitario `enviarDatos` (líneas 146 a 159) para enviar la cadena al servidor. El método `enviarDatos` escribe el objeto, vacía el búfer de salida y anexa la misma cadena al área de texto en la ventana del cliente. Una vez más, no es necesario invocar al método utilitario `mostrarMensaje` para modificar el área de texto aquí, ya que el método `enviarDatos` es llamado desde un manejador de eventos.

18.7 Interacción entre cliente/servidor sin conexión mediante datagramas

Hasta este punto hemos hablado sobre la *transmisión basada en flujos, orientada a la conexión*. Ahora consideremos la *transmisión sin conexión mediante datagramas*. La transmisión orientada a la conexión es como el sistema telefónico en el que usted marca y recibe una *conexión* al teléfono de la persona con la que usted desea comunicarse. La conexión se mantiene todo el tiempo que dure su llamada telefónica, incluso aunque usted no esté hablando.

La transmisión sin conexión mediante *datagramas* es un proceso más parecido a la manera en que el correo se transporta mediante el servicio postal. Si un mensaje extenso no cabe en un sobre, usted lo divide en varias piezas separadas que coloca en sobres separados, numerados en forma secuencial. Cada una de las cartas se envía entonces por correo al mismo tiempo. Las cartas podrían llegar en orden, sin orden o tal vez no llegarían (aunque el último caso es raro, suele ocurrir). La persona en el extremo receptor debe reensamblar las piezas del mensaje en orden secuencial, antes de tratar de interpretarlo. Si su mensaje es lo suficientemente pequeño como para caber en un sobre, no tiene que preocuparse por el problema de que el mensaje esté "fuera de secuencia", pero aún existe la posibilidad de que su mensaje no llegue. Una diferencia entre los datagramas y el correo postal es que pueden llegar duplicados de datagramas al equipo receptor.

En las figuras 18.6 y 18.7 utilizamos datagramas para enviar paquetes de información mediante el Protocolo de datagramas de usuario (*UDP*) entre una aplicación cliente y una aplicación servidor. En la aplicación `Cliente` (figura 18.7), el usuario escribe un mensaje en un campo de texto y oprime *Intro*. El programa convierte el mensaje en un arreglo `byte` y lo coloca en un paquete de datagramas que se envía al servidor. El `Servidor` (figura 18.6) recibe el paquete y muestra la información que contiene, después lo *repite* de vuelta al cliente. Cuando el cliente recibe el paquete, muestra la información que contiene. En este ejemplo, las clases `Cliente` y `Servidor` se implementan de manera similar.

La clase Servidor

La clase `Servidor` (figura 18.6) declara dos objetos `DatagramPacket` que son utilizados por el servidor para enviar y recibir información, y un objeto `DatagramSocket` que envía y recibe estos paquetes. El constructor de `Servidor` (líneas 14 a 35) crea la interfaz gráfica de usuario en la que se mostrarán los paquetes de información. A continuación, en la línea 26 se crea el objeto `DatagramSocket` en un bloque `try`; y además se utiliza el constructor de `DatagramSocket` que toma un argumento entero para el número de puerto (`5000` en este ejemplo) para enlazar el servidor a un puerto en donde pueda recibir paquetes de los clientes. Los objetos `Cliente` que envían paquetes a este objeto `Servidor` especifican el mismo número de puerto en los paquetes que envían. Si el constructor de `DatagramSocket` no puede enlazar el objeto `DatagramSocket` al puerto especificado, se lanza una excepción `SocketException`.



Error común de programación 18.2

Al especificar un puerto que ya esté en uso, o especificar un número de puerto incorrecto al crear un objeto `DatagramSocket`, se produce una excepción `SocketException`.

http://librosysolucionarios.net

El método **esperarPaquetes** (líneas 38 a 71) de la clase **Servidor** utiliza un ciclo infinito para esperar a que lleguen los paquetes al **Servidor**. En las líneas 47 y 48 se crea un objeto **DatagramPacket**, en el cual puede almacenarse un paquete de información que se haya recibido. El constructor de **DatagramPacket** para este fin recibe dos argumentos: un arreglo **byte** en el cual se almacenarán los datos y la longitud del arreglo. En la línea 50 se utiliza el método ***receive*** de **DatagramSocket** para esperar a que llegue un paquete al **Servidor**. El método **receive** hace un bloqueo hasta que llega el paquete y después lo almacena en su argumento **DatagramPacket**. Este método lanza una excepción **IOException** si ocurre un error al recibir un paquete.

```
1  // Fig. 18.6: Servidor.java
2  // Servidor que recibe y envía paquetes de/a un cliente.
3  import java.io.*;
4  import java.net.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class Servidor extends JFrame {
10    private JTextArea areaPantalla;
11    private DatagramSocket socket;
12
13    // configurar la GUI y DatagramSocket
14    public Servidor()
15    {
16       super( "Servidor" );
17
18       areaPantalla = new JTextArea();
19       getContentPane().add( new JScrollPane( areaPantalla ),
20          BorderLayout.CENTER );
21       setSize( 400, 300 );
22       setVisible( true );
23
24       // crear objeto DatagramSocket para enviar y recibir paquetes
25       try {
26          socket = new DatagramSocket( 5000 );
27       }
28
29       // procesar los problemas que pueden ocurrir al crear el objeto DatagramSocket
30       catch( SocketException excepcionSocket ) {
31          excepcionSocket.printStackTrace();
32          System.exit( 1 );
33       }
34
35    } // fin del constructor de Servidor
36
37    // esperar a que lleguen los paquetes, mostrar los datos y repetir el paquete al cliente
38    private void esperarPaquetes()
39    {
40       while ( true ) { // iterar infinitamente
41
42          // recibir paquete, mostrar su contenido, devolver copia al cliente
43          try {
44
```

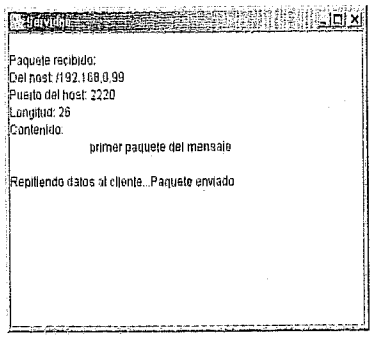
Figura 18.6 Lado servidor de la computación cliente/servidor sin conexión mediante datagramas. (Parte 1 de 3.)

http://librosysolucionarios.net


```
45          // establecer el paquete
46          byte datos[] = new byte[ 100 ];
47          DatagramPacket recibirPaquete =
48             new DatagramPacket( datos, datos.length );
49
50          socket.receive( recibirPaquete ); // esperar el paquete
51
52          // mostrar la información del paquete recibido
53          mostrarMensaje( "\nPaquete recibido:" +
54             "\nDel host: " + recibirPaquete.getAddress() +
55             "\nPuerto del host: " + recibirPaquete.getPort() +
56             "\nLongitud: " + recibirPaquete.getLength() +
57             "\nContenido:\n\t" + new String( recibirPaquete.getData(),
58                0, recibirPaquete.getLength() ) );
59
60          enviarPaqueteACliente( recibirPaquete ); // enviar paquete al cliente
61       }
62
63       // procesar los problemas que pueden ocurrir al manipular el paquete
64       catch( IOException excepcionES ) {
65          mostrarMensaje( excepcionES.toString() + "\n" );
66          excepcionES.printStackTrace();
67       }
68
69    } // fin de instrucción while
70
71 } // fin del método esperarPaquetes
72
73 // repetir el paquete al cliente
74 private void enviarPaqueteACliente( DatagramPacket recibirPaquete )
75    throws IOException
76 {
77    mostrarMensaje( "\n\nRepitiendo datos al cliente..." );
78
79    // crear paquete a enviar
80    DatagramPacket enviarPaquete = new DatagramPacket(
81       recibirPaquete.getData(), recibirPaquete.getLength(),
82       recibirPaquete.getAddress(), recibirPaquete.getPort() );
83
84    socket.send( enviarPaquete ); // enviar el paquete
85    mostrarMensaje( "Paquete enviado\n" );
86 }
87
88 // método utilitario que es llamado desde otros subprocesos para manipular a
89 // areaPantalla en el subproceso despachador de eventos
90 private void mostrarMensaje( final String mensajeAMostrar )
91 {
92    // mostrar el mensaje del subproceso de ejecución despachador de eventos
93    SwingUtilities.invokeLater(
94       new Runnable() { // clase interna para asegurar que la GUI se actualice
          apropiadamente
95
96          public void run() // actualiza areaPantalla
97          {
98             areaPantalla.append( mensajeAMostrar );
```

Figura 18.6 Lado servidor de la computación cliente/servidor sin conexión mediante datagramas. (Parte 2 de 3.)

```
99                areaPantalla.setCaretPosition(
100                   areaPantalla.getText().length() );
101            }
102
103         } // fin de la clase interna
104
105      ); // fin de la llamada a SwingUtilities.invokeLater
106   }
107
108   public static void main( String args[] )
109   {
110      Servidor aplicacion = new Servidor();
111      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
112      aplicacion.esperarPaquetes();
113   }
114
115 } // fin de la clase Servidor
```



Ventana de **Servidor** después de que el **Cliente** recibe el paquete

Figura 18.6 Lado servidor de la computación cliente/servidor sin conexión mediante datagramas. (Parte 3 de 3.)

Cuando llega un paquete, en las líneas 53 a 58 se hace una llamada al método **mostrarMensaje** (declarado en las líneas 90 a 106) para anexar el contenido del paquete al área de texto. El método ***getAddress*** de **DatagramPacket** (línea 54) devuelve un objeto **InetAddress** que contiene el nombre de host del equipo desde el que se envió el paquete. El método ***getPort*** (línea 55) devuelve un entero que especifica el número de puerto a través del que el equipo host enviará el paquete. El método ***getLength*** (línea 56) devuelve un entero que representa el número de bytes de datos que se enviaron. El método ***getData*** (línea 57) devuelve un arreglo **byte** que contiene los datos que se enviaron. El programa utiliza el arreglo **byte** para inicializar un objeto **String**, el cual se anexa al texto que se mostrará en pantalla.

Después de mostrar un paquete, en la línea 60 se hace una llamada al método **enviarPaqueteACliente** (declarado en las líneas 74 a 86) para crear un nuevo paquete y enviarlo al cliente. En las líneas 80 a 82 se crea un objeto **DatagramPacket** y se le pasan cuatro argumentos a su constructor. El primer argumento especifica el arreglo **byte** a enviar; el segundo especifica el número de bytes a enviar; el tercero, la dirección de Internet del equipo cliente a donde se va a enviar el paquete; y, finalmente, el cuarto argumento especifica el puerto en el que el cliente espera recibir los paquetes. En la línea 83 se envía el paquete a través de la red. El método ***send*** lanza una excepción **IOException** si ocurre un error al enviar un paquete.

La clase Cliente

La clase **Cliente** (figura 18.7) funciona de manera similar a la clase **Servidor**, excepto que **Cliente** envía paquetes sólo cuando el usuario escribe un mensaje en un campo de texto y oprime la tecla *Intro*. Cuando

http://librosysolucionarios.net


```
1  // Fig. 18.7: Cliente.java
2  // Cliente que envía y recibe paquetes a/de un servidor.
3  import java.io.*;
4  import java.net.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class Cliente extends JFrame {
10    private JTextField campoIntroducir;
11    private JTextArea areaPantalla;
12    private DatagramSocket socket;
13
14    // configurar la GUI y DatagramSocket
15    public Cliente()
16    {
17       super( "Cliente" );
18
19       Container contenedor = getContentPane();
20
21       campoIntroducir = new JTextField( "Escriba aquí el mensaje" );
22       campoIntroducir.addActionListener(
23          new ActionListener() {
24             public void actionPerformed( ActionEvent evento )
25             {
26                // crear y enviar el paquete
27                try {
28                   areaPantalla.append( "\nEnviando paquete que contiene: " +
29                      evento.getActionCommand() + "\n" );
30
31                   // obtener mensaje del campo de texto y convertirlo en arreglo byte
32                   String mensaje = evento.getActionCommand();
33                   byte datos[] = mensaje.getBytes();
34
35                   // crear enviarPaquete
36                   DatagramPacket enviarPaquete = new DatagramPacket( datos,
37                      datos.length, InetAddress.getLocalHost(), 5000 );
38
39                   socket.send( enviarPaquete ); // enviar paquete
40                   areaPantalla.append( "Paquete enviado\n" );
41                   areaPantalla.setCaretPosition(
42                      areaPantalla.getText().length() );
43                }
44
45                // procesar los problemas que pueden ocurrir al crear o enviar el paquete
46                catch ( IOException excepcionES ) {
47                   mostrarMensaje( excepcionES.toString() + "\n" );
48                   excepcionES.printStackTrace();
49                }
50
51             } // fin de actionPerformed
52
53          } // fin de la clase interna
54
```

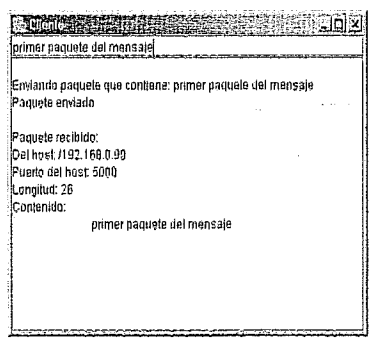
Figura 18.7 Lado cliente de la computación cliente/servidor sin conexión mediante datagramas. (Parte 1 de 3.)

```
55        ); // fin de la llamada a addActionListener
56
57        contenedor.add( campoIntroducir, BorderLayout.NORTH );
58
59        areaPantalla = new JTextArea();
60        contenedor.add( new JScrollPane( areaPantalla ),
61           BorderLayout.CENTER );
62
63        setSize( 400, 300 );
64        setVisible( true );
65
66        // crear objeto DatagramSocket para enviar y recibir paquetes
67        try {
68           socket = new DatagramSocket();
69        }
70
71       // atrapar los problemas que pueden ocurrir al crear objeto DatagramSocket
72       catch( SocketException excepcionSocket ) {
73           excepcionSocket.printStackTrace();
74           System.exit( 1 );
75        }
76
77     } // fin del constructor de Cliente
78
79     // esperar a que lleguen los paquetes del Servidor, mostrar el contenido de los paquetes
80     private void esperarPaquetes()
81     {
82        while ( true ) { // iterar infinitamente
83
84           // recibir el paquete y mostrar su contenido
85           try {
86
87              // establecer el paquete
88              byte datos[] = new byte[ 100 ];
89              DatagramPacket recibirPaquete = new DatagramPacket(
90                 datos, datos.length );
91
92              socket.receive( recibirPaquete ); // esperar un paquete
93
94              // mostrar el contenido del paquete
95              mostrarMensaje( "\nPaquete recibido:" +
96                 "\nDel host: " + recibirPaquete.getAddress() +
97                 "\nPuerto del host: " + recibirPaquete.getPort() +
98                 "\nLongitud: " + recibirPaquete.getLength() +
99                 "\nContenido:\n\t" + new String( recibirPaquete.getData(),
100                   0, recibirPaquete.getLength() ) );
101          }
102
103          // procesar los problemas que pueden ocurrir al recibir o mostrar el paquete
104          catch( IOException excepcion ) {
105             mostrarMensaje( excepcion.toString() + "\n" );
106             excepcion.printStackTrace();
107          }
108
```

Figura 18.7 Lado cliente de la computación cliente/servidor sin conexión mediante datagramas. (Parte 2 de 3.)

http://librosysolucionarios.net


```
109       } // fin de instrucción while
110
111    } // fin del método esperarPaquetes
112
113    // método utilitario que es llamado desde otros subprocesos para manipular a
114    // areaPantalla en el subproceso despachador de eventos
115    private void mostrarMensaje( final String mensajeAMostrar )
116    {
117       // mostrar mensaje del subproceso de ejecución despachador de eventos
118       SwingUtilities.invokeLater(
119          new Runnable() { // clase interna para asegurar que la GUI se actualice
             apropiadamente
120
121             public void run() // actualiza areaPantalla
122             {
123                areaPantalla.append( mensajeAMostrar );
124                areaPantalla.setCaretPosition(
125                   areaPantalla.getText().length() );
126             }
127
128          } // fin de la clase interna
129
130       ); // fin de la llamada a SwingUtilities.invokeLater
131    }
132
133    public static void main( String args[] )
134    {
135       Cliente aplicacion = new Cliente();
136       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
137       aplicacion.esperarPaquetes();
138    }
139
140 } // fin de la clase Cliente
```



Ventana de **Cliente** después de enviar el paquete a **Servidor** y de recibir el paquete de vuelta por parte de **Servidor**

Figura 18.7 Lado cliente de la computación cliente/servidor sin conexión mediante datagramas. (Parte 3 de 3.)

esto ocurre, el programa llama al método **actionPerformed** (líneas 24 a 51), el cual convierte la cadena que introdujo el usuario en un arreglo **byte** (línea 33). En las líneas 36 y 37 se crea un objeto **DatagramPacket** y se inicializa con el arreglo **byte**, la longitud de la cadena introducida por el usuario, la dirección IP a la que se enviará el paquete [**InetAddress.getLocalHost()** en este ejemplo] y el número de puerto en el que el **Servidor** espera los paquetes (5000 en este ejemplo). En la línea 39 se envía el paquete.

Observe que el cliente en este ejemplo debe saber que el servidor está recibiendo paquetes en el puerto 5000; de no ser así, el servidor no los recibirá.

Observe que la llamada al constructor de **DatagramSocket** (línea 68) en esta aplicación no especifica ningún argumento. Este constructor permite a la computadora seleccionar el siguiente número de puerto disponible para el objeto **DatagramSocket**. El cliente no necesita un número de puerto específico, ya que el servidor recibe el número de puerto del cliente como parte de cada objeto **DatagramPacket** enviado por el cliente. Por lo tanto, el servidor puede enviar paquetes de vuelta al mismo equipo y número de puerto desde el cual haya recibido un paquete de información.

El método **esperarPaquetes** (líneas 80 a 111) de la clase **Cliente** utiliza un ciclo infinito para esperar a que lleguen los paquetes del servidor. En la línea 92 se hace un bloqueo hasta que llegue un paquete. Observe que esto no impide al usuario enviar un paquete, ya que los eventos de la GUI se manejan en el subproceso despachador de eventos. Sólo impide que el ciclo **while** continúe ejecutándose, hasta que llegue un paquete al **Cliente**. Cuando llega un paquete, en la línea 92 se almacena este paquete en recibirPaquete, y en las líneas 95 a 100 se hace una llamada al método **mostrarMensaje** (declarado en las líneas 115 a 131) para mostrar el contenido del paquete en el área de texto.

18.8 Juego de tres en raya (gato) tipo cliente/servidor, utilizando un servidor con subprocesamiento múltiple

En esta sección presentaremos el popular juego de Tres en raya (Gato o Tic-Tac-Toe), utilizando técnicas cliente/servidor con sockets de flujo. El programa consiste en una aplicación **TresEnRaya** (figura 18.8) que permite a dos applet **ClienteTresEnRaya** (figura 18.9) conectarse al servidor y jugar Tres en raya. En la figura 18.10 se muestran pantallas de ejemplo de la salida de este programa.

La clase ***ServidorTresEnRaya***

A medida que el **ServidorTresEnRaya** recibe la conexión de cada cliente, crea una instancia de la clase interna **Jugador** (líneas 163 a 280 de la figura 18.8) para procesar al cliente en un subproceso separado. Estos subprocesos permiten a los clientes jugar en forma independiente. El primer cliente que se conecta al servidor es el jugador X y el segundo cliente que se conecta es el jugador O. El jugador X realiza el primer movimiento. El servidor mantiene la información acerca del tablero, para poder determinar si el movimiento de un jugador es válido o inválido.

```
1  // Fig. 18.8: ServidorTresEnRaya.java
2  // Esta clase mantiene un juego de Tres en raya para dos applet cliente.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.net.*;
6  import java.io.*;
7  import javax.swing.*;
8
9  public class ServidorTresEnRaya extends JFrame {
10    private char[] tablero;
11    private JTextArea areaSalida;
12    private Jugador[] jugadores;
13    private ServerSocket servidor;
14    private int jugadorActual;
15    private final int JUGADOR_X = 0, JUGADOR_O = 1;
16    private final char MARCA_X = 'X', MARCA_O = 'O';
17
18    // establecer servidor de tres en raya y la GUI para mostrar mensajes
19    public ServidorTresEnRaya()
20    {
21       super( "Servidor de Tres en raya" );
```

Figura 18.8 Lado servidor del programa cliente/Servidor TresEnRaya. (Parte 1 de 6.)


```
22
23       tablero = new char[ 9 ];
24       jugadores = new Jugador[ 2 ];
25       jugadorActual = JUGADOR_X;
26
27       // establecer objeto ServerSocket
28       try {
29          servidor = new ServerSocket( 12345, 2 );
30       }
31
32       // procesar los problemas que pueden ocurrir al crear el objeto ServerSocket
33       catch( IOException excepcionES ) {
34          excepcionES.printStackTrace();
35          System.exit( 1 );
36       }
37
38       // establecer objeto JTextArea para mostrar mensajes durante la ejecución
39       areaSalida = new JTextArea();
40       getContentPane().add( areaSalida, BorderLayout.CENTER );
41       areaSalida.setText( "Servidor esperando conexiones\n" );
42
43       setSize( 300, 300 );
44       setVisible( true );
45
46    } // fin del constructor de ServidorTresEnRaya
47
48    // esperar dos conexiones para poder jugar
49    public void ejecutar()
50    {
51       // esperar a que se conecte cada cliente
52       for ( int i = 0; i < jugadores.length; i++ ) {
53
54          // esperar conexión, crear Jugador, iniciar subproceso
55          try {
56             jugadores[ i ] = new Jugador( servidor.accept(), i );
57             jugadores[ i ].start();
58          }
59
60          // procesar los problemas que pueden ocurrir al recibir la conexión del cliente
61          catch( IOException excepcionES ) {
62             excepcionES.printStackTrace();
63             System.exit( 1 );
64          }
65       }
66
67       // El Jugador X se suspende hasta que se conecte el Jugador O.
68       // Reactivar ahora al jugador X.
69       synchronized ( jugadores[ JUGADOR_X ] ) {
70          jugadores[ JUGADOR_X ].establecerSuspendido( false );
71          jugadores[ JUGADOR_X ].notify();
72       }
73
74    } // fin del método ejecutar
75
76    // método utilitario que es llamado desde otros subprocesos para manipular a
```

Figura 18.8 Lado servidor del programa cliente/Servidor TresEnRaya. (Parte 2 de 6.)

http://librosysolucionarios.net

```
77    // areaSalida en el subproceso despachador de eventos
78    private void mostrarMensaje( final String mensajeAMostrar )
79    {
80       // mostrar mensaje del subproceso de ejecución despachador de eventos
81       SwingUtilities.invokeLater(
82          new Runnable() {  // clase interna para asegurar
83          que la GUI se actualice apropiadamente
84             public void run() // actualiza a areaSalida
85             {
86                areaSalida.append( mensajeAMostrar );
87                areaSalida.setCaretPosition(
88                   areaSalida.getText().length() );
89             }
90
91          }  // fin de la clase interna
92
93       ); // fin de la llamada a SwingUtilities.invokeLater
94    }
95
96    // Determinar si un movimiento es válido. Este método es sincronizado porque
97    // sólo puede realizarse un movimiento a la vez.
98    public synchronized boolean validarYMover( int posicion, int jugador )
99    {
100      boolean movimientoRealizado = false;
101
102      // mientras no sea el jugador actual, debe esperar su turno
103      while ( jugador != jugadorActual ) {
104
105         // esperar su turno
106         try {
107            wait();
108         }
109
110         // atrapar interrupciones de wait
111         catch( InterruptedException excepcionInterrupcion ) {
112            excepcionInterrupcion.printStackTrace();
113         }
114      }
115
116      // si la posición no está ocupada, realizar movimiento
117      if ( !estaOcupada( posicion ) ) {
118
119         // establecer movimiento en arreglo del tablero
120         tablero[ posicion ] = jugadorActual == JUGADOR_X ? MARCA_X : MARCA_O;
121
122         // cambiar jugador actual
123         jugadorActual = ( jugadorActual + 1 ) % 2;
124
125         // hacer saber al nuevo jugador actual que ocurrió un movimiento
126         jugadores[ jugadorActual ].elOtroJugadorMovio( posicion );
127
128         notify(); // indicar al jugador en espera que continúe
129
130         // indicar al jugador que hizo el movimiento, que éste fue válido
131         return true;
```

http://librosysolucionarios.net

Figura 18.8 Lado servidor del programa cliente/Servidor TresEnRaya. (Parte 3 de 6.)


```
132      }
133
134      // indicar al jugador que hizo el movimiento, que éste no fue válido
135      else
136         return false;
137
138   } // fin del método validarYMover
139
140   // determinar si la posición está ocupada
141   public boolean estaOcupada( int posicion )
142   {
143      if ( tablero[ posicion ] == MARCA_X || tablero [ posicion ] == MARCA_O )
144         return true;
145      else
146         return false;
147   }
148
149   // colocar código en este método para determinar si terminó el juego
150   public boolean terminoElJuego()
151   {
152      return false;  // éste se deja como un ejercicio
153   }
154
155   public static void main( String args[] )
156   {
157      ServidorTresEnRaya aplicacion = new ServidorTresEnRaya();
158      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
159      aplicacion.ejecutar();
160   }
161
162   // la clase interna privada Jugador administra a cada Jugador como un subproceso
163   private class Jugador extends Thread {
164      private Socket conexion;
165      private DataInputStream entrada;
166      private DataOutputStream salida;
167      private int numeroJugador;
168      private char marca;
169      protected boolean suspendido = true;
170
171      // establecer subproceso para Jugador
172      public Jugador( Socket socket, int numero )
173      {
174         numeroJugador = numero;
175
176         // especificar la marca del jugador
177         marca = ( numeroJugador == JUGADOR_X ? MARCA_X : MARCA_O );
178
179         conexion = socket;
180
181         // obtener flujos del  objeto Socket
182         try {
183            entrada = new DataInputStream( conexion.getInputStream() );
184            salida = new DataOutputStream( conexion.getOutputStream() );
185         }
186
```

Figura 18.8 Lado servidor del programa cliente/Servidor TresEnRaya. (Parte 4 de 6.)

```
187          // procesar los problemas que pueden ocurrir al obtener los flujos
188          catch( IOException excepcionES ) {
189             excepcionES.printStackTrace();
190             System.exit( 1 );
191          }
192
193       } // fin del constructor de Jugador
194
195       // enviar mensaje indicando que el otro jugador hizo un movimiento
196       public void elOtroJugadorMovio( int posicion )
197       {
198          // enviar mensaje indicando el movimiento
199          try {
200             salida.writeUTF( "El oponente hizo un movimiento" );
201             salida.writeInt( posicion );
202          }
203
204          // procesar los problemas que pueden ocurrir al enviar un mensaje
205          catch ( IOException excepcionES ) {
206             excepcionES.printStackTrace();
207          }
208       }
209
210       // controlar la ejecución del subproceso
211       public void run()
212       {
213          // enviar mensaje al cliente indicando su marca (X u O),
214          // procesar mensajes del cliente
215          try {
216             mostrarMensaje( "Jugador " + ( numeroJugador ==
217                JUGADOR_X ? MARCA_X : MARCA_O ) + " conectado\n" );
218
219             salida.writeChar( marca ); // enviar marca del jugador
220
221             // enviar mensaje indicando que hay conexión
222             salida.writeUTF( "Jugador " + ( numeroJugador == JUGADOR_X ?
223                "X conectado\n" : "O conectado, por favor espere\n" ) );
224
225             // si es jugador X, esperar a que llegue el otro jugador
226             if ( marca == MARCA_X ) {
227                salida.writeUTF( "Esperando al otro jugador" );
228
229                // esperando al jugador O
230                try {
231                   synchronized( this ) {
232                      while ( suspendido )
233                         wait();
234                   }
235                }
236
237                // procesar interrupciones mientras está en espera
238                catch ( InterruptedException excepcion ) {
239                   excepcion.printStackTrace();
240                }
241
```

http://librosysolucionarios.net

Figura 18.8 Lado servidor del programa cliente/Servidor TresEnRaya. (Parte 5 de 6.)


```
242             // enviar mensaje indicando que el otro jugador se conectó y
243             // que el jugador X puede realizar un movimiento
244             salida.writeUTF( "El otro jugador se conectó. Le toca a usted mover." );
245          }
246
247          // mientras el juego no esté terminado
248          while ( ! terminoElJuego() ) {
249
250             // obtener del cliente la posición del movimiento
251             int posicion = entrada.readInt();
252
253             // comprobar que sea un movimiento válido
254             if ( validarYMover( posicion, numeroJugador ) ) {
255                mostrarMensaje( "\nposición: " + posicion );
256                salida.writeUTF( "Movimiento válido." );
257             }
258             else
259                salida.writeUTF( "Movimiento inválido, intente otra vez" );
260          }
261
262          conexion.close(); // cerrar conexión con el cliente
263
264       } // fin del bloque try
265
266       // procesar los problemas que pueden ocurrir al comunicarse con el cliente
267       catch( IOException excepcionES ) {
268          excepcionES.printStackTrace();
269          System.exit( 1 );
270       }
271
272    } // fin del método run
273
274    // establecer si el subproceso está suspendido o no
275    public void establecerSuspendido( boolean estado )
276    {
277       suspendido = estado;
278    }
279
280 } // fin de la clase Jugador
281
282 } // fin de la clase ServidorTresEnRaya
```

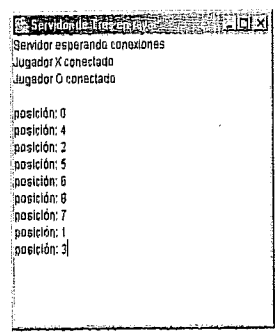


Figura 18.8 Lado servidor del programa cliente/Servidor TresEnRaya. (Parte 6 de 6.)

Empezaremos con una discusión sobre el lado servidor del juego de Tres en raya. Al ejecutarse la aplicación `ServidorTresEnRaya`, el método `main` (líneas 155 a 160) crea un objeto `ServidorTresEnRaya` llamado `aplicacion`. El constructor (líneas 19 a 46) trata de configurar un objeto `ServerSocket`. Si tiene éxito, el programa muestra la ventana del servidor y después `main` invoca al método `ejecutar` (líneas 49 a 74) de `ServidorTresEnRaya`. El método `ejecutar` itera dos veces, haciendo un bloqueo en la línea 56 cada vez que espera la conexión de un cliente. Cuando un cliente se conecta, en la línea 56 se crea un nuevo objeto `Jugador` para administrar la conexión como un subproceso separado, y en la línea 57 se hace una llamada al método `start` de ese objeto, para empezar a ejecutar el subproceso.

Cuando el `ServidorTresEnRaya` crea a un `Jugador`, el constructor de `Jugador` (líneas 172 a 193) recibe el objeto `Socket` que representa la conexión con el cliente y obtiene los flujos de entrada y salida asociados. El método `run` de `Jugador` (líneas 211 a 272) controla la información que se envía al cliente y la información que se recibe del cliente. Primero, pasa al cliente el carácter que el cliente va a colocar en el tablero cuando se haga un movimiento (línea 219). Después indica al cliente que se ha realizado la conexión (líneas 222 y 223). En las líneas 231 a 234 se suspende el subproceso del jugador X cuando empieza a ejecutarse, ya que el jugador X podrá realizar movimientos sólo hasta después de que el jugador O se conecte.

En este punto se puede empezar a jugar, y el método `run` empieza a ejecutar su estructura `while` (líneas 248 a 260). En cada iteración de este ciclo se lee un entero (línea 250) que representa la posición en donde el cliente desea colocar una marca, y en la línea 254 se invoca al método `validarYMover` (declarado en las líneas 98 a 138) de `ServidorTresEnRaya` para comprobar el movimiento. Si el movimiento es válido, en la línea 256 se envía un mensaje al cliente, indicando que el movimiento fue válido; en caso contrario, en la línea 259 se envía un mensaje al cliente indicando que el movimiento fue inválido. El programa mantiene las posiciones en el tablero como números del 0 al 8 (del 0 al 2 para la primera fila, del 3 al 5 para la segunda y del 6 al 8 para la tercera).

El método `validarYMover` (líneas 98 a 138 en la clase `ServidorTresEnRaya`) es un método `synchronized`, el cual permite que se mueva solamente un jugador en un momento dado. La sincronización de `validarYMover` evita que ambos jugadores modifiquen la información de estado del juego al mismo tiempo. Si el `Jugador` que trata de validar un movimiento no es el jugador actual (es decir, el que puede hacer un movimiento), ese `Jugador` se coloca en estado de *espera* hasta que sea su turno para realizar un movimiento. Si la posición para el movimiento que se está validando ya está ocupada en el tablero, `validarYMover` devuelve `false`. En caso contrario, el servidor coloca una marca para el jugador en su representación local del tablero (línea 120), notifica al otro objeto `Jugador` (línea 126) que se ha realizado un movimiento (para que se pueda enviar un mensaje al cliente), invoca al método `notify` (línea 128) para que el `Jugador` en espera (si hay uno) pueda validar un movimiento y devuelve `true` (línea 131) para indicar que el movimiento es válido.

La clase de applet `ClienteTresEnRaya`

Cada applet `ClienteTresEnRaya` (figura 18.9) mantiene su propia versión de GUI del tablero de Tres en raya en el que se muestra el estado del juego. Los clientes pueden colocar una marca solamente en un cuadro vacío en el tablero. La clase interna `Cuadro` (líneas 226 a 281 de la figura 18.9) implementa a cada uno de los nueve cuadros en el tablero. Cuando un applet `ClienteTresEnRaya` comienza a ejecutarse, crea un objeto `JTextArea` en el cúal se muestran los mensajes del servidor, junto con una representación del tablero en la que se utilizan nueve objetos `Cuadro`. El método `start` del applet (líneas 67 a 89) abre una conexión con el servidor y obtiene los flujos de entrada y salida asociados del objeto `Socket`. En la línea 73 se realiza una conexión con el host del applet. El nombre del host se obtiene llamando al método `getCodeBase` de `JApplet` para obtener el URL del applet y después llamar al método `getHost` de `URL` para obtener el nombre de host del URL. La clase `ClienteTresEnRaya` implementa a la interfaz `Runnable`, por lo que un subproceso separado puede leer los mensajes del servidor. Este método permite al usuario interactuar con el tablero (en el subproceso despachador de eventos) mientras espera mensajes del servidor. Después de establecer la conexión con el servidor, en la línea 86 se crea el objeto `Thread` llamado `subprocesoSalida`, el cual se inicializa con este (`this`) applet (un objeto `Runnable`), y después en la línea 88 se hace una llamada al método `start` del subproceso. El método `run` del applet (líneas 92 a 122) controla el subproceso de ejecución. Primero, el método lee el carácter de la marca (X u O) del servidor (línea 96), después itera continuamente (líneas

111 a 113) y lee los mensajes del servidor (línea 112). Cada mensaje se pasa al método `procesarMensaje` del applet (líneas 125 a 166) para su procesamiento.

Si el mensaje que se recibe es `"Movimiento válido."`, en las líneas 129 y 130 se muestra el mensaje `"Movimiento válido, por favor espere."` y se hace una llamada al método `establecerMarca` (líneas 189 a 199) para establecer la marca del cliente en el cuadro actual (el cuadro en el que el usuario hizo clic), utilizando el método `invokeLater` de `SwingUtilities` para asegurar que las actualizaciones de la GUI ocurran en el subproceso despachador de eventos. Si el mensaje recibido es `"Movimiento inválido, intente de nuevo."` en la línea 135 se muestra el mensaje para que el usuario pueda hacer clic en un cuadro distinto. Si el mensaje recibido es `"El oponente realizó un movimiento."`, en las líneas 144 a 151 se lee un entero del servidor indicando a qué lugar se movió el oponente, y se coloca una marca en ese cuadro del tablero (de nuevo utilizando al método `invokeLater` de `SwingUtilities` para asegurar que las actualizaciones en la GUI ocurran en el subproceso despachador de eventos). Si se recibe cualquier otro mensaje, en la línea 164 simplemente se muestra ese mensaje en pantalla. En la figura 18.10 se muestran capturas de pantalla de dos applet interactuando mediante el `ServidorTresEnRaya` como ejemplo.

```
1  // Fig. 18.9: ClienteTresEnRaya.java
2  // Cliente que permite a un usuario jugar Tres en raya con otro usuario, a través de una red.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.net.*;
6  import java.io.*;
7  import javax.swing.*;
8
9  public class ClienteTresEnRaya extends JApplet implements Runnable {
10    private JTextField campoID;
11    private JTextArea areaPantalla;
12    private JPanel panelTablero, panel2;
13    private Cuadro tablero[][], cuadroActual;
14    private Socket conexion;
15    private DataInputStream entrada;
16    private DataOutputStream salida;
17    private char miMarca;
18    private boolean miTurno;
19    private final char MARCA_X = 'X', MARCA_O = 'O';
20
21    // establecer interfaz de usuario y tablero
22    public void init()
23    {
24       Container contenedor = getContentPane();
25
26       // establecer objeto JTextArea para mostrar mensajes al usuario
27       areaPantalla = new JTextArea( 4, 30 );
28       areaPantalla.setEditable( false );
29       contenedor.add( new JScrollPane( areaPantalla ), BorderLayout.SOUTH );
30
31       // establecer panel para los cuadros en el tablero
32       panelTablero = new JPanel();
33       panelTablero.setLayout( new GridLayout( 3, 3, 0, 0 ) );
34
35       // crear tablero
36       tablero = new Cuadro[ 3 ][ 3 ];
37
```

Figura 18.9 Lado cliente del programa cliente/Servidor TresEnRaya. (Parte 1 de 6.)

http://librosysolucionarios.net

```
38       // Al crear un objeto Cuadro, el argumento posicion para el constructor
39       // es un valor entre 0 y 8, indicando la posición del objeto Cuadro en
40       // el tablero. Los valores 0,  y 2 son la primera fila, los valores 3, 4
41       // y 5 son la segunda fila. Los valores 6, 7 y 8 son la tercera fila.
42       for ( int fila = 0; fila < tablero.length; fila++ ) {
43
44          for ( int columna = 0; columna < tablero[ fila ].length; columna++ ) {
45
46             // crear objeto Cuadro
47             tablero[ fila ][ columna ] = new Cuadro( ' ', fila * 3 + columna );
48             panelTablero.add( tablero[ fila ][ columna ] );
49          }
50       }
51
52       // campo de texto para mostrar la marca del jugador
53       campoID = new JTextField();
54       campoID.setEditable( false );
55       contenedor.add( campoID, BorderLayout.NORTH );
56
57       // establecer panel para contener a panelTablero (para su distribución en
            la pantalla)
58       panel2 = new JPanel();
59       panel2.add( panelTablero, BorderLayout.CENTER );
60       contenedor.add( panel2, BorderLayout.CENTER );
61
62    } // fin del método init
63
64    // Realizar conexión con el servidor y obtener flujos asociados.
65    // Iniciar subproceso separado para permitir a este subprograma
66    // actualizar continuamente su salida en el área de texto de la pantalla.
67    public void start()
68    {
69       // conectarse con el servidor, obtener los flujos e iniciar subprocesoSalida
70       try {
71
72          // realizar la conexión
73          conexion = new Socket( getCodeBase().getHost(), 12345 );
74
75          // obtener los flujos
76          entrada = new DataInputStream( conexion.getInputStream() );
77          salida = new DataOutputStream( conexion.getOutputStream() );
78       }
79
80       // atrapar los problemas que pueden ocurrir al establecer la conexión y los flujos
81       catch ( IOException excepcionES ) {
82          excepcionES.printStackTrace();
83       }
84
85       // crear e iniciar subproceso de salida
86       Thread subprocesoSalida = new Thread( this );
87       subprocesoSalida.start();
88
89    } // fin del método start
90
91    // controlar subproceso que permite actualización continua de areaPantalla
92    public void run()
```

Figura 18.9 Lado cliente del programa cliente/Servidor TresEnRaya. (Parte 2 de 6.)


```
93    {
94       // obtener la marca del jugador (O u X)
95       try {
96          miMarca = entrada.readChar();
97
98          // mostrar ID del jugador en subproceso despachador de eventos
99          SwingUtilities.invokeLater(
100            new Runnable() {
101               public void run()
102               {
103                  campoID.setText( "Usted es el jugador \"" + miMarca + "\"" );
104               }
105            }
106         );
107
108         miTurno = ( miMarca == MARCA_X ? true : false );
109
110         // recibir mensajes enviados al cliente y mostrarlos en pantalla
111         while ( true ) {
112            procesarMensaje( entrada.readUTF() );
113         }
114
115      } // fin del bloque try
116
117      // procesar los problemas que pueden ocurrir al comunicarse con el servidor
118      catch ( IOException excepcionES ) {
119         excepcionES.printStackTrace();
120      }
121
122   } // fin del método run
123
124   // procesar los mensajes recibidos por el cliente
125   private void procesarMensaje( String mensaje )
126   {
127      // ocurrió un movimiento válido
128      if ( mensaje.equals( "Movimiento válido." ) ) {
129         mostrarMensaje( "Movimiento válido, por favor espere.\n" );
130         establecerMarca( cuadroActual, miMarca );
131      }
132
133      // ocurrió un movimiento inválido
134      else if ( mensaje.equals( "Movimiento inválido, intente otra vez" ) ) {
135         mostrarMensaje( mensaje + "\n" );
136         miTurno = true;
137      }
138
139      // el oponente realizó un movimiento
140      else if ( mensaje.equals( "El oponente hizo un movimiento" ) ) {
141
142         // obtener posición del movimiento y actualizar el tablero
143         try {
144            int posicion = entrada.readInt();
145            int fila = posicion / 3;
146            int columna = posicion % 3;
```

Figura 18.9 Lado cliente del programa cliente/Servidor TresEnRaya. (Parte 3 de 6.)

http://librosysolucionarios.net

```
147
148                establecerMarca(  tablero[ fila ][ columna ],
149                   ( miMarca == MARCA_X ? MARCA_O : MARCA_X ) );
150                mostrarMensaje( "El oponente hizo un movimiento. Su turno.\n" );
151                miTurno = true;
152
153             } // fin del bloque try
154
155             // procesar los problemas que pueden ocurrir al comunicarse con el servidor
156             catch ( IOException excepcionES ) {
157                excepcionES.printStackTrace();
158             }
159
160          } // fin de la instrucción else if
161
162          // mostrar simplemente el mensaje
163          else
164             mostrarMensaje( mensaje + "\n" );
165
166       } // fin del método procesarMensaje
167
168       // método utilitario que es llamado desde otros subprocesos para manipular a
169       // areaSalida en el subproceso despachador de eventos
170       private void mostrarMensaje( final String mensajeAMostrar )
171       {
172          // mostrar mensaje del subproceso de ejecución despachador de eventos
173          SwingUtilities.invokeLater(
174             new Runnable() { // clase interna para asegurar que la GUI
175             se actualice apropiadamente
176                public void run() // actualiza areaPantalla
177                {
178                   areaPantalla.append( mensajeAMostrar );
179                   areaPantalla.setCaretPosition(
180                      areaPantalla.getText().length() );
181                }
182
183             } // fin de la clase interna
184
185          ); // fin de la llamada a SwingUtilities.invokeLater
186       }
187
188       // método utilitario para establecer la marca en el tablero, en el subproceso
          despachador de eventos
189       private void establecerMarca( final Cuadro cuadroAMarcar, final char marca )
190       {
191          SwingUtilities.invokeLater(
192             new Runnable() {
193                public void run()
194                {
195                   cuadroAMarcar.establecerMarca( marca );
196                }
197             }
198          );
199       }
200
201       // enviar mensaje al servidor indicando el cuadro en el
```

http://librosysolucionarios.net

Figura 18.9 Lado cliente del programa cliente/servidor TresEnRaya. (Parte 4 de 6.)


```
202       public void enviarCuadroClic( int posicion )
203       {
204          if ( miTurno ) {
205
206             // enviar posicion al servidor
207             try {
208                salida.writeInt( posicion );
209                miTurno = false;
210             }
211
212             // procesar los problemas que pueden ocurrir al comunicarse con el servidor
213             catch ( IOException excepcionES ) {
214                excepcionES.printStackTrace();
215             }
216          }
217       }
218
219       // establecer el cuadro actual
220       public void establecerCuadroActual( Cuadro cuadro )
221       {
222          cuadroActual = cuadro;
223       }
224
225       // clase interna privada para los cuadros en el tablero
226       private class Cuadro extends JPanel {
227          private char marca;
228          private int posicion;
229
230          public Cuadro( char marcaCuadro, int posicionCuadro )
231          {
232             marca = marcaCuadro;
233             posicion = posicionCuadro;
234
235             addMouseListener(
236                new MouseAdapter() {
237                   public void mouseReleased( MouseEvent e )
238                   {
239                      establecerCuadroActual( Cuadro.this );
240                      enviarCuadroClic( obtenerPosicionCuadro() );
241                   }
242                }
243             );
244
245          } // fin del constructor de Cuadro
246
247          // devolver tamaño preferido del Cuadro
248          public Dimension getPreferredSize()
249          {
250             return new Dimension( 30, 30 );
251          }
252
253          // devolver tamaño mínimo del Cuadro
254          public Dimension getMinimumSize()
255          {
256             return getPreferredSize();
```

Figura 18.9 Lado cliente del programa cliente/Servidor TresEnRaya. (Parte 5 de 6.)

```
257         }
258
259         // establecer marca para Cuadro
260         public void establecerMarca( char nuevaMarca )
261         {
262            marca = nuevaMarca;
263            repaint();
264         }
265
266         // devolver posición del Cuadro
267         public int obtenerPosicionCuadro()
268         {
269            return posicion;
270         }
271
272         // dibujar el Cuadro
273         public void paintComponent( Graphics g )
274         {
275            super.paintComponent( g );
276
277            g.drawRect( 0, 0, 29, 29 );
278            g.drawString( String.valueOf( marca ), 11, 20 );
279         }
280
281      } // fin de la clase interna Cuadro
282
283   } // fin de la clase ClienteTresEnRaya
```

Figura 18.9 Lado cliente del programa cliente/servidor TresEnRaya. (Parte 6 de 6.)

18.9 La seguridad y la red

A pesar de que sería muy interesante escribir una gran variedad de poderosas aplicaciones basadas en red, nuestros esfuerzos podrían estar limitados debido a cuestiones relacionadas con la seguridad. Muchos navegadores Web tales como Netscape y Microsoft Internet Explorer prohíben, de manera predeterminada, que los applet realicen un procesamiento de los archivos en los equipos en los que se ejecutan. Piense en ello. Un applet de Java está diseñado para ser enviado a su navegador a través de un documento HTML que puede descargarse de cualquier servidor Web en el mundo. Por lo general, es muy poco lo que podrá saber acerca de los orígenes de los

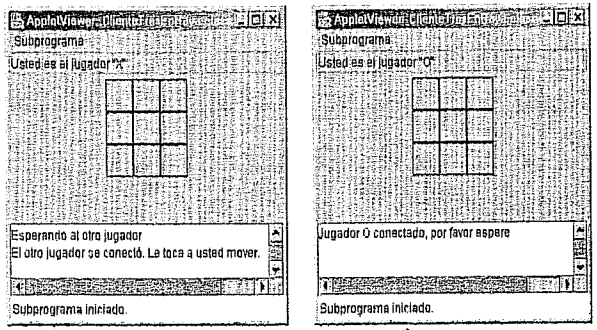


Figura 18.10 Resultados de ejemplo del programa cliente/servidor TresEnRaya. (Parte 1 de 2.)

http://librosysolucionarios.net

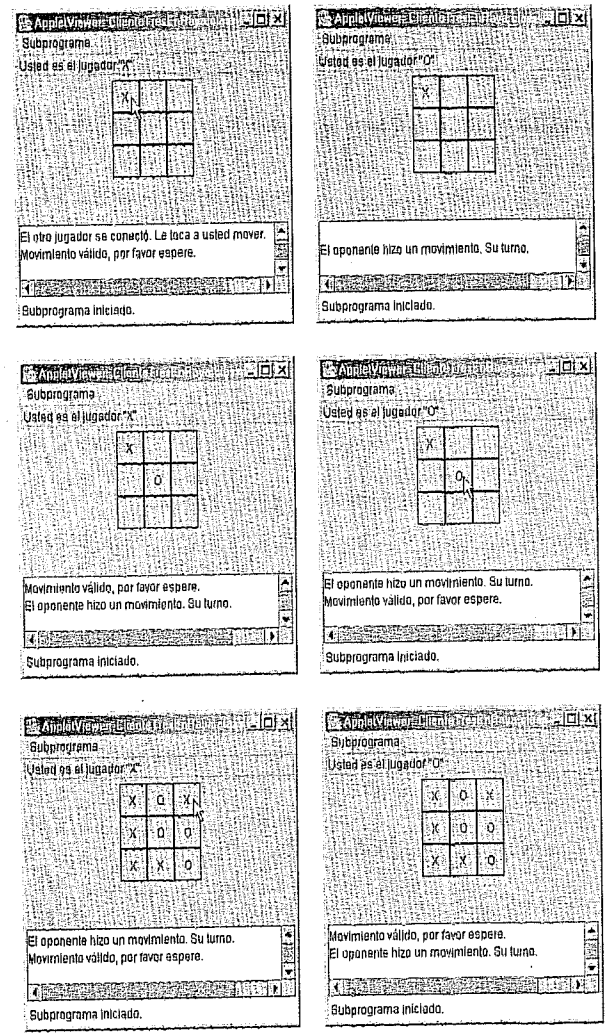


Figura 18.10 Resultados de ejemplo del programa cliente/servidor TresEnRaya. (Parte 2 de 2.)

applet de Java que se ejecuten en su sistema. Podría ser desastroso permitir que estos applet tuvieran la libertad de manipular sus archivos.

Una situación más sutil ocurre al limitar los equipos con los que los applet en ejecución pueden realizar conexiones de red. Para crear aplicaciones con una verdadera capacidad de colaboración, sería ideal que nuestros applet pudieran comunicarse con equipos en casi cualquier parte. Por lo general, el administrador de seguridad de Java en un navegador Web restringe a un applet de manera que sólo pueda comunicarse con el equipo desde el que se descargó originalmente.

Estas restricciones podrían parecer demasiado rígidas. Sin embargo, la API de seguridad de Java ahora proporciona herramientas para applet con firma digital, con lo cual los navegadores podrán determinar si un applet se descarga desde un *origen de confianza (trusted source)*. En casos en los que un applet es de confianza, éste

puede recibir acceso adicional al equipo en el que se está ejecutando. Las características de la API de seguridad de Java y ciertas herramientas de red adicionales se describen en nuestro texto *Advanced Java 2 Platform How to Program*.

18.10 Servidor y cliente para conversaciones DeitelMessenger

Los salones de conversación se han vuelto bastante comunes en Internet dado que proporcionan una ubicación central en donde los usuarios pueden conversar entre sí, mediante mensajes de texto cortos. Cada participante en un salón de conversación puede ver todos los mensajes que publican los demás usuarios, y cada uno de los usuarios puede publicar mensajes en el salón de conversación. En esta sección presentaremos nuestro ejemplo práctico culminante sobre redes, el cual integra muchas de las características de Java relacionadas con redes, subprocesamiento múltiple y la GUI de Swing que hemos aprendido hasta ahora, para crear un sistema de conversación en línea. También presentaremos la *transmisión múltiple (multicast)*, que permite a una aplicación enviar objetos `DatagramPacket` a grupos de clientes. Después de leer esta sección, usted podrá crear aplicaciones de red más considerables.

18.10.1 `ServidorDeitelMessenger` y las clases de soporte

`ServidorDeitelMessenger` (figura 18.11) es el corazón del sistema de conversación en línea. Esta clase aparece en el paquete `com.deitel.messenger.sockets.servidor`. Los clientes pueden participar en una conversación conectándose al objeto `ServidorDeitelMessenger`. El método `iniciarServidor` (líneas 16 a 51) inicia el `ServidorDeitelMessenger`. En las líneas 22 y 23 se crea un objeto `ServerSocket` para aceptar conexiones de red entrantes. Recuerde que el constructor de `ServerSocket` toma como su primer argumento el puerto en el que el servidor debe escuchar las conexiones entrantes. La interfaz `ConstantesSocketMessenger` (figura 18.12) declara el número de puerto como la constante `PUERTO_SERVIDOR`, para asegurar que tanto el servidor como los clientes utilicen el número de puerto correcto.

En las líneas 29 a 42 se escucha continuamente, en espera de nuevas conexiones de clientes. En la línea 32 se invoca el método `accept` de `ServerSocket` para esperar y aceptar una nueva conexión de cliente. En la línea 36 se crea y se inicia un nuevo objeto `SubprocesoReceptor` para el cliente. La clase `SubprocesoReceptor` (figura 18.14) del paquete `com.deitel.messenger.sockets.servidor` es una sub-

```
1  // Fig. 18.11: ServidorDeitelMessenger.java
2  // ServidorDeitelMessenger es un servidor de conversación (chat) con
3  // subprocesamiento múltiple, basado en sockets y paquetes.
4  package com.deitel.messenger.sockets.servidor;
5
6  import java.util.*;
7  import java.net.*;
8  import java.io.*;
9
10 import com.deitel.messenger.*;
11 import com.deitel.messenger.sockets.*;
12
13 public class ServidorDeitelMessenger implements EscuchaMensajes {
14
15    // iniciar servidor de conversación
16    public void iniciarServidor()
17    {
18       // crear servidor y administrar nuevos clientes
19       try {
20
21          // crear objeto ServerSocket para las conexiones entrantes
22          ServerSocket socketServidor = new ServerSocket(
23             ConstantesSocketMessenger.PUERTO_SERVIDOR, 100 );
24
25          System.out.println( "Servidor escuchando en el puerto " +
26             ConstantesSocketMessenger.PUERTO_SERVIDOR + " ..." );
27
28          // escuchar constantemente en espera de clientes
29          while ( true ) {
30
31             // aceptar nueva conexión de cliente
32             Socket socketCliente = socketServidor.accept();
33
34             // crear nuevo objeto SubprocesoReceptor para recibir
35             // mensajes del cliente
36             new SubprocesoReceptor( this, socketCliente ).start();
37
38             // imprimir información sobre la conexión
39             System.out.println( "Conexion recibida de: " +
40                socketCliente.getInetAddress() );
41
42          } // fin de instrucción while
43
44       } // fin del bloque try
45
46       // manejar posible excepción al crear servidor y conectar clientes
47       catch ( IOException excepcionES ) {
48          excepcionES.printStackTrace();
49       }
50
51    } // fin del método iniciarServidor
52
53    // al recibir un nuevo mensaje, transmitirlo a los clientes
54    public void mensajeRecibido( String de, String mensaje )
55    {
56       // crear objeto String que contenga el mensaje completo
57       String mensajeCompleto = de +
58          ConstantesSocketMessenger.SEPARADOR_MENSAJE + mensaje;
59
60       // crear e iniciar objeto SubprocesoEnvioMulticast para transmitir
61       // los nuevos mensajes a todos los clientes
62       new SubprocesoEnvioMulticast( mensajeCompleto.getBytes() ).start();
63    }
64
65    public static void main ( String args[] )
66    {
67       new ServidorDeitelMessenger().iniciarServidor();
68    }
69
70 } // fin de la clase ServidorDeitelMessenger
```

```
Servidor escuchando en el puerto 12345 ...
Conexion recibida de: SEANSANTRY/XXX.XXX.XXX.XXX
Conexion recibida de: PJD/XXX.XXX.XXX.XXX
```

http://librosysolucionarios.net

Figura 18.11 `ServidorDeitelMessenger` para administrar un salón de conversación. (Parte 1 de 2.)

Figura 18.11 `ServidorDeitelMessenger` para administrar un salón de conversación. (Parte 2 de 2.)

clase de **Thread** que escucha los mensajes entrantes que provienen de un cliente. El primer argumento para el constructor de **SubprocesoReceptor** es un objeto **EscuchaMensajes** (figura 18.13), al cual se deben entregar los mensajes provenientes del cliente. La clase **ServidorDeitelMessenger** implementa a la interfaz **EscuchaMensajes** (línea 13) del paquete **com.deitel.messenger**, y por lo tanto puede pasarle la referencia **this** al constructor de **SubprocesoReceptor**.

Cuando cada objeto **SubprocesoReceptor** recibe un nuevo mensaje de un cliente, el objeto **SubprocesoReceptor** pasa el mensaje a un objeto **EscuchaMensajes** a través del método **mensajeRecibido** (líneas 54 a 63). En las líneas 57 y 58 se concatena la cadena **de** con el separador **>>>** y el cuerpo del mensaje. En la línea 62 se crea y se inicia un nuevo objeto **SubprocesoEnvioMulticast** para entregar el **mensajeCompleto** a todos los clientes. La clase **SubprocesoEnvioMulticast** (figura 18.15) del paquete **com.deitel.messenger.sockets.servidor** utiliza la *transmisión múltiple (multicast)* como un mecanismo eficiente para enviar un mensaje a varios clientes. Hablaremos en breve acerca de los detalles de la transmisión múltiple. En el método **main** (líneas 65 a 68) se crea una nueva instancia de **ServidorDeitelMessenger** y se inicia el servidor.

La interfaz **ConstantesSocketMessenger** (figura 18.12) declara las constantes a usar en las diversas clases que conforman el sistema de mensajería Deitel. Las clases pueden acceder a estas constantes estáticas haciendo referencia a ellas mediante la interfaz **ConstantesSocketMessenger** (por ejemplo, **ConstantesSocketMessenger.PUERTO_SERVIDOR**).

En la línea 9 se declara la constante **String**, llamada **DIRECCION_MULTICAST**, que contiene la dirección a la cual un objeto **SubprocesoEnvioMulticast** (figura 18.15) debe enviar mensajes. Ésta es una de las direcciones reservadas para la transmisión múltiple, que describiremos en la discusión de la figura 18.15. En la línea 12 se declara la constante entera **PUERTO_ESCUCHA_MULTICAST**: el puerto en el que los clientes deben escuchar los nuevos mensajes. En la línea 15 se declara la constante entera **PUERTO_ENVIO_MULTICAST**: el puerto al cual un objeto **SubprocesoEnvioMulticast** debe publicar nuevos mensajes en la **DIRECCION_MULTICAST**. En la línea 18 se declara la constante entera **PUERTO_SERVIDOR**: el puerto en el que **ServidorDeitelMessenger** escucha las conexiones entrantes de los clientes. En la línea 21 se declara la constante **String** llamada **CADENA_DESCONEXION**, que es el objeto **String** que envía un cliente al objeto **ServidorDeitelMessenger** cuando el usuario desea salir del salón de conversación. En la línea 24 se declara la constante **String** llamada **SEPARADOR_MENSAJE**, la cual separa el nombre del usuario del cuerpo del mensaje. En la línea 27 se especifica el tamaño máximo de los mensajes en bytes.

```
1  // Fig. 18.12: ConstantesSocketMessenger.java
2  // ConstantesSocketMessenger define las constantes para los números de puerto
3  // y dirección de transmisión múltiple en DeitelMessenger
4  package com.deitel.messenger.sockets;
5
6  public interface ConstantesSocketMessenger {
7
8     // dirección para transmisión múltiple de datagramas
9     public static final String DIRECCION_MULTICAST = "239.0.0.1";
10
11    // puerto para escuchar transmisión múltiple de datagramas
12    public static final int PUERTO_ESCUCHA_MULTICAST = 5555;
13
14    // puerto para enviar datagramas de transmisión múltiple
15    public static final int PUERTO_ENVIO_MULTICAST = 5554;
16
17    // puerto para conexiones de objetos Socket a ServidorDeitelMessenger
18    public static final int PUERTO_SERVIDOR = 5000;
19
20    // Objeto String que indica desconexión
21    public static final String CADENA_DESCONEXION = "DESCONECTAR";
22
23    // Objeto String que separa el nombre del usuario del cuerpo del mensaje
24    public static final String SEPARADOR_MENSAJE = ">>>";
25
26    // tamaño del mensaje (en bytes)
27    public static final int TAMANIO_MENSAJE = 512;
28 }
```

Figura 18.12 La clase **ConstantesSocketMessenger** declara las constantes a utilizar en las clases **ServidorDeitelMessenger** y **DeitelMessenger**.

http://librosysolucionarios.net


```
1  // Fig. 18.13: EscuchaMensajes.java
2  // EscuchaMensajes es una interfaz para las clases que desean
3  // recibir nuevos mensajes de conversación.
4  package com.deitel.messenger;
5
6  public interface EscuchaMensajes {
7
8     // recibir nuevo mensaje de conversación
9     public void mensajeRecibido( String de, String mensaje);
10 }
```

Figura 18.13 La interfaz **EscuchaMensajes** que declara el método **mensajeRecibido** para recibir nuevos mensajes de conversación.

ServidorDeitelMessenger utiliza instancias de la clase **SubprocesoReceptor** (figura 18.14) del paquete **com.deitel.messenger.sockets.servidor**, para escuchar nuevos mensajes de cada uno de los clientes. La clase **SubprocesoReceptor** extiende a la clase **Thread**. Esto permite a **ServidorDeitelMessenger** crear un objeto de la clase **SubprocesoReceptor** para cada cliente, para manejar mensajes de varios clientes a la vez. Cuando **ServidorDeitelMessenger** recibe una nueva conexión de un cliente, crea un nuevo objeto **SubprocesoReceptor** para ese cliente y después continúa escuchando a la espera de nuevas conexiones de clientes. El objeto **SubprocesoReceptor** escucha los mensajes de un solo cliente y los regresa a **ServidorDeitelMessenger**, a través del método **mensajeRecibido**.

```
1  // Fig. 18.14: SubprocesoReceptor.java
2  // SubprocesoReceptor es un objeto Thread que escucha mensajes de un
3  // cliente específico y entrega mensajes a un objeto EscuchaMensajes.
4  package com.deitel.messenger.sockets.servidor;
5
6  import java.io.*;
7  import java.net.*;
8  import java.util.StringTokenizer;
9
10 import com.deitel.messenger.*;
11 import com.deitel.messenger.sockets.*;
12
13 public class SubprocesoReceptor extends Thread {
```

Figura 18.14 La clase **SubprocesoReceptor** para escuchar los nuevos mensajes de los clientes de **ServidorDeitelMessenger** en subprocesos separados. (Parte 1 de 3.)

```
14
15    private BufferedReader entrada;
16    private EscuchaMensajes escuchaMensajes;
17    private boolean seguirEscuchando = true;
18
19    // constructor de SubprocesoReceptor
20    public SubprocesoReceptor( EscuchaMensajes escucha, Socket socketCliente )
21    {
22       // invocar al constructor de la superclase para nombrar objeto Thread
23       super( "SubprocesoReceptor: " + socketCliente );
24
25       // establecer escucha al que deben enviarse los nuevos mensajes
26       escuchaMensajes = escucha;
27
28       // establecer tiempo de interrupción para leer de socketCliente y crear
29       // objeto BufferedReader para leer mensajes entrantes
30       try {
31          socketCliente.setSoTimeout( 5000 );
32
33          entrada = new BufferedReader( new InputStreamReader(
34             socketCliente.getInputStream() ) );
35       }
36
37       // manejar posible excepción al crear objeto BufferedReader
38       catch ( IOException excepcionES ) {
39          excepcionES.printStackTrace();
40       }
41
42    } // fin del constructor de SubprocesoReceptor
43
44    // escuchar nuevos mensajes y entregarlos a EscuchaMensajes
45    public void run()
46    {
47       String mensaje;
48
49       // escuchar mensajes hasta ser detenido
50       while ( seguirEscuchando ) {
51
52          // leer mensaje de objeto BufferedReader
53          try {
54             mensaje = entrada.readLine();
55          }
56
57          // manejar excepción si se acaba el tiempo para la lectura
58          catch ( InterruptedIOException excepcionInterrupcion ) {
59
60             // continuar con la siguiente iteración para seguir escuchando
61             continue;
62          }
63
64          // manejar excepción al leer mensaje
65          catch ( IOException excepcionES ) {
66             excepcionES.printStackTrace();
67             break;
```

Figura 18.14 La clase **SubprocesoReceptor** para escuchar los nuevos mensajes de los clientes de **ServidorDeitelMessenger** en subprocesos separados. (Parte 2 de 3.)


```
68          }
69
70          // asegurar que el mensaje no sea nulo
71          if ( mensaje != null ) {
72
73             // dividir mensaje en tokens para recuperar nombre de usuario y
74             // cuerpo del mensaje
75             StringTokenizer tokenizer = new StringTokenizer(
76                mensaje, ConstantesSocketMessenger.SEPARADOR_MENSAJE );
77
78             // ignorar mensajes que no contienen un nombre
79             // de usuario y cuerpo del mensaje
80             if ( tokenizer.countTokens() == 2 )
81
82                // enviar mensaje a EscuchaMensajes
83                escuchaMensajes.mensajeRecibido(
84                   tokenizer.nextToken(),    // nombre de usuario
85                   tokenizer.nextToken() ); // cuerpo del mensaje
86
87             else
88
89                // si se recibe mensaje de desconexión, dejar de escuchar
90                if ( mensaje.equalsIgnoreCase(
91                   ConstantesSocketMessenger.SEPARADOR_MENSAJE +
92                   ConstantesSocketMesaenger.CADENA_DESCONEXION ) )
93                   dejarDeEscuchar();
94
95          } // fin de instrucción if
96
97       } // fin de instrucción while
98
99       // cerrar BufferedReader (también se cierra Socket)
100      try {
101         entrada.close();
102      }
103
104      // manejar posible excepción al cerrar BufferedReader
105      catch ( IOException excepcionES ) {
106         excepcionES.printStackTrace();
107      }
108
109   } // fin del método run
110
111   // dejar de escuchar mensajes entrantes
112   public void dejarDeEscuchar()
113   {
114      seguirEscuchando = false;
115   }
116
117 } // fin de la clase SubprocesoReceptor
```

Figura 18.14 La clase **SubprocesoReceptor** para escuchar los nuevos mensajes de los clientes de **ServidorDeitelMessenger** en subprocesos separados. (Parte 3 de 3.).

El constructor de **SubprocesoReceptor** (líneas 20 a 42) toma un objeto **EscuchaMensajes** como su único argumento. El objeto **SubprocesoReceptor** entregará los nuevos mensajes a este componente de escucha, invocando a su método **mensajeRecibido**. El argumento **Sockat** del constructor de **Subproce-**

http://librosysolucionarios.net

`soReceptor` es la conexión a un cliente específico. En la línea 23 se invoca al constructor de `Thread` para proporcionar un nombre único para cada instancia de `SubprocesoReceptor`. Recuerde que nombrar cada subproceso puede ser útil al depurar una aplicación. En la línea 26 se establece el componente `EscuchaMensajes` al que el objeto `SubprocesoReceptor` debe entregar los nuevos mensajes. En la línea 31 se invoca al método *`setSoTimeout`* de la clase `Socket` con un argumento de `5000` milisegundos. El proceso de leer datos de un objeto `Socket` es una *llamada de bloqueo*: el subproceso actual se coloca en estado bloqueado (figura 16.1) mientras el subproceso espera a que la operación de lectura se complete. El método `setSoTimeout` especifica que, si no se reciben datos en el número dado de milisegundos, el objeto `Socket` debe producir una excepción `InterruptedIOException`, la cual puede ser atrapada por el subproceso actual y luego seguir con la ejecución. Esta técnica evita que el subproceso actual sufra un interbloqueo si no hay más datos disponibles que provengan del objeto `Socket`. En las líneas 33 y 34 se crea un nuevo objeto `BufferedReader` para el objeto `InputStream` de `socketCliente`. El objeto `SubprocesoReceptor` utiliza este objeto `BufferedReader` para leer los nuevos mensajes del cliente.

El método `run` (líneas 45 a 108) escucha continuamente, a la espera de nuevos mensajes por parte del cliente. En las líneas 50 a 96 se ejecuta el ciclo, siempre y cuando la variable `boolean` llamada `seguirEscuchando` sea `true`. En la línea 54 se invoca el método `readLine` de `BufferedReader` para leer una línea de texto del cliente. Si pasan más de `5000` milisegundos sin leer datos, el método `readLine` lanza una excepción `InterruptedIOException`, la cual indica que ha expirado el tiempo de interrupción establecido en la línea 31. En la línea 61 se utiliza una instrucción `continue` para avanzar a la siguiente iteración del ciclo `while` y continuar escuchando mensajes. En las líneas 65 a 68 se atrapa una excepción `IOException`, la cual indica un problema más severo, proveniente del método `readLine`. En este caso, en la línea 66 se imprime un rastreo de la pila para ayudar a depurar la aplicación, y en la línea 67 se utiliza la palabra clave `break` para terminar el ciclo.

Cuando un cliente envía un mensaje al servidor, el cliente separa el nombre del usuario del cuerpo del mensaje con la constante `ConstantesSocketMessenger.SEPARADOR_MENSAJE`. Si no hay excepciones lanzadas al leer datos del cliente, y el mensaje no es `null` (línea 71), en las líneas 74 y 75 se crea un nuevo objeto `StringTokenizer` que utiliza el delimitador `ConstantesSocketMessenger.SEPARADOR_MENSAJE` para separar cada mensaje en dos tokens: el nombre de usuario del emisor y el mensaje. En la línea 79 se comprueba que haya el número apropiado de tokens, y en las líneas 82 a 84 se invoca el método `mensajeRecibido` de la interfaz `EscuchaMensajes` para entregar el nuevo mensaje al objeto `EscuchaMensajes` registrado. Si el objeto `StringTokenizer` no produce dos tokens, en las líneas 89 a 91 se comprueba el mensaje para ver si concuerda con la constante `ConstantesSocketMessenger.CADENA_DESCONEXION`. Si las cadenas concuerdan, en la línea 92 se invoca el método `dejarDeEscuchar` de `SubprocesoReceptor` para terminar este subproceso.

El método `dejarDeEscuchar` (líneas 111 a 114) establece la variable `boolean` llamada `seguirEscuchando` en `false`. Esto ocasiona que la condición del ciclo `while` que empieza en la línea 50 falle, y también ocasiona que el objeto `SubprocesoReceptor` cierre el objeto `Socket` del cliente (línea 100). Después, el método `run` regresa, el cual termina la ejecución del objeto `SubprocesoReceptor`.

La clase `SubprocesoEnvioMulticast` (figura 18.15) entrega objetos `DatagramPacket`, que contienen mensajes de conversación, a un grupo de clientes. La transmisión múltiple es una manera efectiva de enviar datos a muchos clientes sin incurrir en la sobrecarga producida al transmitir esos datos a todos los hosts en Internet. Para comprender el concepto de transmisión múltiple, veamos una analogía del mundo real: la relación entre una editorial de una revista y los suscriptores de esa revista. La editorial de revistas produce una revista y la proporciona a un distribuidor. Los clientes interesados en esa revista obtienen una suscripción y empiezan a recibir la revista en el correo, proveniente del distribuidor. Este proceso de comunicación es bastante distinto al de una transmisión de televisión. Cuando una estación de televisión produce un programa de televisión, la estación transmite ese programa a lo largo de una región geográfica, o tal vez en todo el mundo mediante el uso de satélites. Transmitir un programa de televisión para 10,000 espectadores no es más costoso para la estación de televisión que transmitir un programa para 100 espectadores; la señal de radio que lleva la transmisión llega a un área amplia. Sin embargo, imprimir y entregar una revista a 10,000 lectores sería un proceso mucho más costoso que imprimir y entregar la revista a 100 lectores. La mayoría de las editoriales de revistas no podrían seguir operando si tuvieran que transmitir sus revistas a todo el mundo, por lo que en vez de eso utilizan la transmisión múltiple para transmitir sus revistas a un grupo de suscriptores.


```
1  // Fig. 18.15: SubprocesoEnvioMulticast.java
2  // SubprocesoEnvioMulticast es un objeto Thread que transmite un mensaje
3  // de conversación utilizando un datagrama de transmisión múltiple.
4  package com.deitel.messenger.sockets.servidor;
5
6  import java.io.*;
7  import java.net.*;
8
9  import com.deitel.messenger.sockets.*;
10
11 public class SubprocesoEnvioMulticast extends Thread {
12
13    // datos del mensaje
14    private byte[] bytesMensaje;
15
16    // constructor de SubprocesoEnvioMulticast
17    public SubprocesoEnvioMulticast( byte[] bytes )
18    {
19       // invocar al constructor de la superclase para nombrar al objeto Thread
20       super( "SubprocesoEnvioMulticast" );
21
22       bytesMensaje = bytes;
23    }
24
25    // entregar mensaje a DIRECCION_MULTICAST a través de objeto DatagramSocket
26    public void run()
27    {
28       // entregar mensaje
29       try {
30
31          // crear objeto DatagramSocket para enviar el mensaje
32          DatagramSocket socket = new DatagramSocket(
33             ConstantesSocketMessenger.PUERTO_ENVIO_MULTICAST );
34
35          // usar objeto InetAddress reservado para grupo de transmisión múltiple
36          InetAddress grupo = InetAddress.getByName(
37             ConstantesSocketMessenger.DIRECCION_MULTICAST );
38
39          // crear objeto DatagramPacket que contiene el mensaje
40          DatagramPacket paquete = new DatagramPacket( bytesMensaje,
41             bytesMensaje.length, grupo,
42             ConstantesSocketMessenger.PUERTO_ESCUCHA_MULTICAST );
43
44          // enviar paquete al grupo de transmisión múltiple y cerrar el socket
45          socket.send( paquete );
46          socket.close();
47       }
48
49       // manejar posible excepción al entregar el mensaje
50       catch ( IOException excepcionES ) {
51          excepcionES.printStackTrace();
52       }
53
54    } // fin del método run
```

Figura 18.15 La clase `SubprocesoEnvioMulticast` para entregar mensajes salientes a un grupo mediante objetos `DatagramPacket`. (Parte 1 de 2.)

http://librosysolucionario.net

```
55
56 } // fin de la clase SubprocesoEnvioMulticast
```

Figura 18.15 La clase `SubprocesoEnvioMulticast` para entregar mensajes salientes a un grupo mediante objetos `DatagramPacket`. (Parte 2 de 2.)

Mediante el uso de la transmisión múltiple, una aplicación puede "publicar" objetos `DatagramPacket` para entregarlos a las aplicaciones "suscriptoras". Una aplicación transmite objetos `DatagramPacket` en forma múltiple, enviando los objetos `DatagramPacket` a una *dirección de transmisión múltiple (multicast)*, la cual es una dirección de IP reservada para este tipo de transmisión. Las direcciones de transmisión múltiple se encuentran en el rango de `224.0.0.0` a `239.255.255.255`. Las direcciones que empiezan con `239` están reservadas para intranets, por lo que utilizaremos una de éstas (`239.0.0.1`) en nuestro ejemplo práctico. Los clientes que desean recibir estos objetos `DatagramPacket` pueden conectarse a la dirección de transmisión múltiple apropiada para unirse al grupo de suscriptores (el *grupo de transmisión múltiple*). Cuando una aplicación envía un objeto `DatagramPacket` a la dirección de transmisión múltiple, cada cliente en el grupo de transmisión múltiple recibe el objeto `DatagramPacket`. A diferencia de los objetos `DatagramPacket` de transmisión única (figura 18.7), los objetos `DatagramPacket` de transmisión múltiple no son confiables; no se garantiza que los paquetes lleguen a cualquier destino. Además, el orden en el que cada uno de los clientes recibe los datagramas no se garantiza.

La clase `SubprocesoEnvioMulticast` extiende a la clase `Thread` para permitir a `ServidorDeitelMessenger` enviar mensajes de transmisión múltiple en un subproceso separado. Para enviar un mensaje de transmisión múltiple, el objeto `ServidorDeitelMessenger` crea un objeto `SubprocesoEnvioMulticast` con el contenido del mensaje e inicia ese subproceso. El constructor de `SubprocesoEnvioMulticast` (líneas 17 a 23) toma como argumento un arreglo de objetos `byte`, los cuales contienen el mensaje.

El método `run` (líneas 26 a 54) entrega el mensaje a la dirección de transmisión múltiple. En las líneas 32 y 33 crea un nuevo objeto `DatagramSocket`. Recuerde que en la sección 18.7 utilizamos objetos `DatagramSocket` para enviar objetos `DatagramPacket` de *transmisión única*; paquetes que se envían directamente de un host a otro. Los objetos `DatagramPacket` de transmisión múltiple se envían de la misma forma, sólo que la dirección a la que se envían los paquetes es una dirección de transmisión múltiple. En las líneas 36 y 37 se crea un objeto `InetAddress` para la dirección de transmisión múltiple, el cual se declara como una constante en la interfaz `ConstantesSocketMessenger`. En las líneas 40 a 42 se crea el objeto `DatagramPacket` que contiene el mensaje. El primer argumento para el constructor de `DatagramPacket` es el arreglo `byte` que contiene el mensaje; el segundo argumento es la longitud del arreglo `byte`; el tercero especifica el objeto `InetAddress` al que debe enviarse el paquete, y el último argumento especifica el número de puerto al cual se debe entregar el paquete en la dirección de transmisión múltiple. En la línea 45 se envía el paquete mediante el método `send` de `DatagramSocket`. Todos los clientes que escuchan la dirección de transmisión múltiple en el puerto apropiado, recibirán este objeto `DatagramPacket`. En la línea 46 se cierra el objeto `DatagramSocket` y el método `run` regresa terminando el objeto `SubprocesoEnvioMulticast`.

Ejecución del `ServidorDeitelMessenger`

Para ejecutar el `ServidorDeitelMessenger`, abra una ventana de comandos y cambie al directorio en donde se encuentra el paquete `com.deitel.messenger.sockets.servidor` (es decir, el directorio en el que se encuentra `com`). Después, escriba:

```
java com.deitel.messenger.sockets.servidor.ServidorDeitelMessenger
```

para ejecutar el servidor.

18.10.2 Cliente `DeitelMessenger` y las clases de soporte

El cliente para el `ServidorDeitelMessenger` tiene varios componentes. Una clase que implementa a la interfaz `AdministradorMensajes` (figura 18.16) administra la comunicación con el servidor. Una subclase de `Thread` escucha los mensajes en la dirección de transmisión múltiple del `ServidorDeitelMessenger`. Otra subclase de `Thread` envía los mensajes del cliente al servidor. Una subclase de `JFrame` proporciona la GUI de los clientes.

http://librosysolucionarios.net


```
1  // Fig. 18.16: AdministradorMensajes.java
2  // AdministradorMensajes es una interfaz para objetos capaz de administrar
3  // las comunicaciones con un servidor de mensajes.
4  package com.deitel.messenger;
5
6  public interface AdministradorMensajes {
7
8     // conectarse al servidor de mensajes y enrutar los mensajes entrantes
9     // al objeto EscuchaMensajes dado
10    public void conectar( EscuchaMensajes escucha );
11
12    // desconectarse del servidor de mensajes y dejar de enrutar
13    // los mensajes entrantes hacia el objeto EscuchaMensajes dado
14    public void desconectar( EscuchaMensajes escucha );
15
16    // enviar mensaje al servidor de mensajes
17    public void enviarMensaje( String de, String mensaje );
18 }
```

Figura 18.16 La interfaz `AdministradorMensajes` que declara los métodos para comunicarse con un objeto `ServidorDeitelMessenger`.

La interfaz `AdministradorMensajes` (figura 18.16) declara métodos para administrar la comunicación con `ServidorDeitelMessenger`. Declaramos esta interfaz para abstraer la funcionalidad básica que necesita un cliente para interactuar con un servidor de conversación (chat), desde el mecanismo de comunicación subyacente que se necesita para comunicarse con ese servidor de conversación. Esta abstracción nos permite proporcionar implementaciones de `AdministradorMensajes` que utilicen otros protocolos de red para implementar los detalles de comunicación. Por ejemplo, si deseamos conectarnos a un servidor de conversación distinto, que no utilice objetos `DatagramPacket` con transmisión múltiple, podríamos implementar la interfaz `AdministradorMensajes` con los protocolos de red apropiados para este servidor de mensajería alternativo. No necesitaríamos modificar ningún código en el cliente, ya que los demás componentes del cliente hacen referencia sólo a la interfaz `AdministradorMensajes`, y no a una implementación específica de `AdministradorMensajes`. De manera similar, los métodos de `AdministradorMensajes` hacen referencia a otros componentes del cliente, sólo a través de la interfaz `EscuchaMensajes`, de manera que otros componentes del cliente pueden cambiar sin necesidad de que haya cambios en el objeto `AdministradorMensajes` ni en sus implementaciones. El método `connect` (línea 10) conecta un objeto `AdministradorMensajes` al `ServidorDeitelMessenger` y enruta los mensajes entrantes al objeto `EscuchaMensajes` apropiado. El método `disconnect` (línea 14) desconecta un objeto `AdministradorMensajes` del `ServidorDeitelMessenger` y deja de entregar mensajes al objeto `EscuchaMensajes` dado. El método `enviarMensaje` (línea 17) envía un nuevo mensaje al `ServidorDeitelMessenger`.

La clase `AdministradorSocketsMensajes` (figura 18.17) implementa a `AdministradorMensajes` (línea 13), utilizando objetos `Socket` y `MulticastSocket` para comunicarse con el `ServidorDeitelMessenger` y recibir los mensajes entrantes. En la línea 16 se declara el objeto `Socket` utilizado para conectarse y enviar mensajes al `ServidorDeitelMessenger`. En la línea 22 se declara un `SubprocesoReceptorPaquetes` (figura 18.19) que escucha los nuevos mensajes entrantes. La bandera `conectado` (línea 25) indica si el objeto `AdministradorSocketsMensajes` se encuentra actualmente conectado al `ServidorDeitelMessenger`.

El constructor de `AdministradorSocketsMensajes` (líneas 28 a 31) recibe la dirección del `ServidorDeitelMessenger` al que debe conectarse el `AdministradorSocketsMensajes`. El método `conectar` (líneas 34 a 59) conecta el objeto `AdministradorSocketsMensajes` al `ServidorDeitelMessenger`. Si `AdministradorSocketsMensajes` estaba ya conectado, en la línea 38 se regresa del método `conectar`. En las líneas 42 y 44 se crea un nuevo objeto `Socket` para comunicarse con el ser-

vidor. En la línea 43 se crea un objeto **InetAddress** para la dirección del servidor, y en la línea 44 se utiliza la constante **ConstantesSocketMessenger.PUERTO_SERVIDOR** para especificar el puerto al que se debe conectar el cliente. En la línea 47 se crea un nuevo objeto **SubprocesoReceptorPaquetes**, el cual escucha los mensajes de transmisión múltiple entrantes que provienen del servidor, y en la línea 48 se inicia el subproceso. En la línea 51 se actualiza la variable **boolean** llamada **conectado** para indicar que el objeto **AdministradorSocketsMensajes** está conectado al servidor.

```
1  // Fig. 18.17: AdministradorSocketsMensajes.java
2  // AdministradorSocketsMensajes se comunica con un ServidorDeitelMessenger usando
3  // objetos Socket y MulticastSocket.
4  package com.deitel.messenger.sockets.cliente;
5
6  import java.util.*;
7  import java.net.*;
8  import java.io.*;
9
10 import com.deitel.messenger.*;
11 import com.deitel.messenger.sockets.*;
12
13 public class AdministradorSocketsMensajes implements AdministradorMensajes {
14
15    // Socket para mensajes salientes
16    private Socket socketCliente;
17
18    // Dirección de ServidorDeitelMessenger
19    private String direccionServidor;
20
21    // Subproceso para recibir mensajes de transmisión múltiple
22    private SubprocesoReceptorPaquetes subprocesoReceptor;
23
24    // bandera indicando el estado de la conexión
25    private boolean conectado = false;
26
27    // constructor de AdministradorSocketsMensajes
28    public AdministradorSocketsMensajes( String direccion )
29    {
30       direccionServidor = direccion;
31    }
32
33    // conectarse al servidor y enviar mensajes al objeto EscuchaMensajes dado
34    public void conectar( EscuchaMensajes escucha )
35    {
36       // si ya está conectado, regresar inmediatamente
37       if ( conectado )
38          return;
39
40       // abrir conexión con objeto Socket a ServidorDeitelMessenger
41       try {
42          socketCliente = new Socket(
43             InetAddress.getByName( direccionServidor ),
44             ConstantesSocketMessenger.PUERTO_SERVIDOR );
45
```

Figura 18.17 La clase **AdministradorSocketsMensajes** es una implementación de la interfaz **AdministradorMensajes** para comunicarse mediante objetos **Socket** y transmitir objetos **DatagramPacket** en forma múltiple. (Parte 1 de 3.)


```
46          // crear objeto Thread para recibir mensajes entrantes
47          subprocesoReceptor = new SubprocesoReceptorPaquetes( escucha );
48          subprocesoReceptor.start();
49
50          // actualizar bandera conectado
51          conectado = true;
52       }
53
54       // manejar posible excepción al conectarse con el servidor
55       catch ( IOException excepcionES ) {
56          excepcionES.printStackTrace();
57       }
58
59    } // fin del método conectar
60
61    // desconectarse del servidor y desregistrar objeto EscuchaMensajes dado
62    public void desconectar( EscuchaMensajes escucha )
63    {
64       // si no está conectado, regresar inmediatamente
65       if ( !conectado )
66          return;
67
68       // detener subproceso de escucha y desconectar del servidor
69       try {
70
71          // notificar al servidor que el cliente se va a desconectar
72          Thread desconectarSubproceso = new SubprocesoEmisor( socketCliente, "",
73             ConstantesSocketMessenger.CADENA_DESCONEXION );
74          desconectarSubproceso.start();
75
76          // esperar 10 segundos para que se envíe el mensaje de desconexión
77          desconectarSubproceso.join( 10000 );
78
79          // detener subprocesoReceptor y eliminar objeto EscuchaMensajes dado
80          subprocesoReceptor.dejarDeEscuchar();
81
82          // cerrar socket saliente
83          socketCliente.close();
84
85       } // fin del bloque try
86
87       // manejar posible excepción al desconectarse del servidor
88       catch ( IOException excepcionES ) {
89          excepcionES.printStackTrace();
90       }
91
92       // manejar posible excepción al unirse a desconectarSubproceso
93       catch ( InterruptedException excepcionInterrupcion ) {
94          excepcionInterrupcion.printStackTrace();
95       }
96
97       // actualizar bandera conectado
98       conectado = false;
```

Figura 18.17 La clase **AdministradorSocketsMensajes** es una implementación de la interfaz **AdministradorMensajes** para comunicarse mediante objetos **Socket** y transmitir objetos **DatagramPacket** en forma múltiple. (Parte 2 de 3.)

http://librosysolucionarios.net

```
99
100    } // fin del método desconectar
101
102    // enviar mensaje al servidor
103    public void enviarMensaje( String de, String mensaje )
104    {
105       // si no está conectado, regresar inmediatamente
106       if ( !conectado )
107          return;
108
109       // crear e iniciar nuevo SubprocesoEmisor para entregar el mensaje
110       new SubprocesoEmisor( socketCliente, de, mensaje ).start();
111    }
112
113 } // fin del método AdministradorSocketsMensajes
```

Figura 18.17 La clase **AdministradorSocketsMensajes** es una implementación de la interfaz **AdministradorMensajes** para comunicarse mediante objetos **Socket** y transmitir objetos **DatagramPacket** en forma múltiple. (Parte 3 de 3.)

El método **desconectar** (líneas 62 a 100) termina la conexión del objeto **AdministradorSocketsMensajes** con el servidor. Si **AdministradorSocketsMensajes** no está conectado, en la línea 66 se regresa del método **desconectar**. En las líneas 72 y 73 se crea un nuevo **SubprocesoEmisor** (figura 18.18) para enviar la constante **ConstantesSocketMessenger.CADENA_DESCONEXION** al **ServidorDeitelMessenger**. La clase **SubprocesoEmisor** entrega un mensaje al **ServidorDeitelMessenger** a través de la conexión **Socket** del objeto **AdministradorSocketsMensajes**. En la línea 74 se inicia el objeto **SubprocesoEmisor** para entregar el mensaje. En la línea 77 se invoca al método **join** de **SubprocesoEmisor** (heredado de **Thread**) para esperar a que se entregue el mensaje de desconexión. El argumento entero **10000** especifica que el subproceso actual debe esperar solamente 10 segundos para unirse (**join**) con el **SubprocesoEmisor** antes de continuar. Una vez que se ha entregado el mensaje de desconexión, en la línea 80 se invoca el método **dejarDeEscuchar** de **SubprocesoReceptorPaquetes** para dejar de recibir mensajes de conversación entrantes. En la línea 83 se cierra la conexión **Socket** con el **ServidorDeitelMessenger**.

El método **enviarMensaje** (líneas 103 a 111) envía un mensaje saliente al servidor. Si **AdministradorSocketsMensajes** no está conectado, en la línea 107 se regresa del método **enviarMensaje**. En la línea 110 se crea y se inicia un nuevo objeto **SubprocesoEmisor** (figura 18.18) para entregar el nuevo mensaje en un subproceso de ejecución separado.

La clase **SubprocesoEmisor** (figura 18.18), que extiende a **Thread**, entrega los mensajes salientes al servidor en un subproceso de ejecución separado. El constructor de **SubprocesoEmisor** (líneas 17 a 27) toma como argumento el objeto **Socket** a través del cual se va a enviar el mensaje, el **nombreUsuario** del cual provino el mensaje, y el mensaje en sí. En las líneas 25 y 26 se concatenan estos argumentos para construir el **mensajeAEnviar**. La constante **ConstantesSocketMessenger.SEPARADOR_MENSAJE** permite que el destinatario del mensaje lo descomponga en dos partes: el nombre de usuario del emisor y el cuerpo del mensaje (mediante el uso de un objeto **StringTokenizer**).

```
1  // Fig. 18.18: SubprocesoEmisor.java
2  // SubprocesoEmisor envía un mensaje al servidor de conversación en un objeto Thread separado.
3  package com.deitel.messenger.sockets.cliente;
4
5  import java.io.*;
6  import java.net.*;
```

Figura 18.18 La clase **SubprocesoEmisor** para entregar los mensajes salientes al **ServidorDeitelMessenger**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
7
8  import com.deitel.messenger.sockets.*;
9
10 public class SubprocesoEmisor extends Thread {
11
12    // objeto Socket a través del cual se va a enviar el mensaje
13    private Socket socketCliente;
14    private String mensajeAEnviar;
15
16    // constructor de SubprocesoEmisor
17    public SubprocesoEmisor( Socket socket, String nombreUsuario, String mensaje )
18    {
19       // invocar al constructor de la superclase para nombrar el objeto Thread
20       super( "SubprocesoEmisor: " + socket );
21
22       socketCliente = socket;
23
24       // construir el mensaje a enviar
25       mensajeAEnviar = nombreUsuario +
26          ConstantesSocketMessenger.SEPARADOR_MENSAJE + mensaje;
27    }
28
29    // enviar mensaje y salir de objeto Thread
30    public void run()
31    {
32       // enviar mensaje y vaciar objeto PrintWriter
33       try {
34          PrintWriter escritor =
35             new PrintWriter( socketCliente.getOutputStream() );
36          escritor.println( mensajeAEnviar );
37          escritor.flush();
38       }
39
40       // manejar posible excepción al enviar el mensaje
41       catch ( IOException excepcionES ) {
42          excepcionES.printStackTrace();
43       }
44    }
45
46 } // fin de la clase SubprocesoEmisor
```

Figura 18.18 La clase **SubprocesoEmisor** para entregar los mensajes salientes al **ServidorDeitelMessenger**. (Parte 2 de 2.)

El método **run** (líneas 30 a 44) entrega el mensaje completo al servidor, utilizando el objeto **Socket** que se proporciona al constructor de **SubprocesoEmisor**. En las líneas 34 y 35 se crea un nuevo objeto **PrintWriter** para el objeto **OutputStream** de **socketCliente**. En la línea 36 se invoca el método **println** de **PrintWriter** para enviar el mensaje. En la línea 37 se invoca el método **flush** de la clase **PrintWriter** para asegurar que el mensaje se envíe inmediatamente. Observe que la clase **SubprocesoEmisor** no cierra el objeto **socketCliente**. La clase **AdministradorSocketsMensajes** utiliza un nuevo objeto de la clase **SubprocesoEmisor** para cada mensaje que envía el cliente, por lo que el objeto **socketCliente** debe permanecer abierto hasta que el usuario se desconecte del **ServidorDeitelMessenger**.

La clase **SubprocesoReceptorPaquetes** (figura 18.19) extiende a la clase **Thread** para permitir a **AdministradorSocketsMensajes** escuchar los mensajes entrantes en un subproceso de ejecución sepa-

rado. En la línea 16 se declara el componente **EscuchaMensajes** al que el objeto **SubprocesoReceptorPaquetes** entregará los mensajes entrantes. En la línea 19 se declara un objeto **MulticastSocket** para recibir objetos **DatagramPacket** de transmisión múltiple. En la línea 22 se declara una referencia **InetAddress** para la dirección de transmisión múltiple en la que el **ServidorDeitelMessenger** publica los nuevos mensajes de conversación. El objeto **MulticastSocket** se conecta a este objeto **InetAddress** para escuchar los mensajes de conversación entrantes.

El constructor de **SubprocesoReceptorPaquetes** (líneas 28 a 56) toma como argumento el objeto **EscuchaMensajes** al que el objeto **SubprocesoReceptorPaquetes** debe entregar los mensajes entrantes. Recuerde que la interfaz **EscuchaMensajes** declara el método **mensajeRecibido**. Cuando el **SubprocesoReceptorPaquetes** recibe un nuevo mensaje de conversación sobre el objeto **MulticastSocket**, el objeto **SubprocesoReceptorPaquetes** invoca a **mensajeRecibido** para entregar el nuevo mensaje al objeto **EscuchaMensajes**.

En las líneas 38 y 39 se crea un nuevo objeto **MulticastSocket** y se pasa al constructor de **MulticastSocket** la constante **PUERTO_ESCUCHA_MULTICAST** de la interfaz **ConstantesSocketMessenger**. Este argumento especifica el puerto en el que el objeto **MulticastSocket** debe escuchar, en espera de los mensajes de conversación entrantes. En las líneas 41 y 42 se crea un objeto **InetAddress** para la constante **ConstantesSocketMessenger.DIRECCION_MULTICAST**, en la que el **ServidorDeitelMessenger** transmite en forma múltiple los nuevos mensajes de conversación. En la línea 45 se invoca el método ***joinGroup*** de **MulticastSocket** para registrar el objeto **MulticastSocket** de manera que reciba los mensajes enviados a la dirección **DIRECCION_MULTICAST**. En la línea 48 se invoca el método **setSoTimeout** de **MulticastSocket** para especificar que, si no se reciben datos en **5000** milisegundos, el objeto **MulticastSocket** debe producir una excepción **InterruptedIOException**, la cual puede ser atrapada por el subproceso actual y luego seguir con la ejecución. Este método evita que **SuprocesoReceptorPaquetes** se bloquee indefinidamente mientras espera los datos entrantes. Además, si nunca se agotara el tiempo para el objeto **MulticastSocket**, el ciclo **while** no podría comprobar la variable **seguirEscuchando** y, por ende, evitar que el objeto **SubprocesoReceptorPaquetes** se detuviera si **seguirEscuchando** fuera **false**.

El método **run** (líneas 59 a 122) escucha los mensajes entrantes de transmisión múltiple. En las 69 y 70 se crea un objeto **DatagramPacket** para almacenar el mensaje entrante. En la línea 74 se invoca el método **receive** de **MulticastSocket** para leer un paquete entrante de la dirección de transmisión múltiple. Si pasan **5000** milisegundos sin recibir un paquete, el método **receive** lanza una excepción **InterruptedIOException**, ya que anteriormente establecimos un tiempo de interrupción de **5000** milisegundos (línea 48). En la línea 80 se utiliza la instrucción **continue** para continuar con la siguiente iteración del ciclo y escuchar los mensajes entrantes. Para las demás excepciones **IOException**, en la línea 87 se utiliza la instrucción **break** para salir del ciclo **while** y terminar el objeto **SubprocesoReceptorPaquetes**.

En la línea 91 se invoca el método **getData** de **DatagramPacket** para recuperar los datos del mensaje. En la línea 94 se invoca el método **trim** de la clase **String** para eliminar el espacio en blanco adicional del final del mensaje. Recuerde que los objetos **DatagramPacket** son de un tamaño fijo (**512** bytes en este ejemplo) por lo que, si el mensaje es más corto que **512** bytes, habrá espacio en blanco adicional después del mensaje. En las líneas 97 y 98 se crea un objeto **StringTokenizer** para separar el cuerpo del mensaje y el nombre del usuario que envió el mensaje. En la línea 102 se comprueba el número correcto de tokens. En las líneas 105 a 107 se invoca el método **mensajeRecibido** de la interfaz **EscuchaMensajes** para entregar el mensaje entrante al objeto **EscuchaMensajes** del objeto **SubprocesoReceptorPaquetes**.

```
1  // Fig. 18.19: SubprocesoReceptorPaquetes.java
2  // SubprocesoReceptorPaquetes escucha los objetos DatagramPacket que contienen
3  // mensajes de un objeto ServidorDeitelMessenger.
4  package com.deitel.messenger.sockets.cliente;
5
```

Figura 18.19 La clase **SubprocesoReceptorPaquetes** para escuchar los nuevos mensajes de transmisión múltiple del **ServidorDeitelMessenger** en un subproceso separado. (Parte 1 de 4.)

http://librosysolucionarios.net


```
6  import java.io.*;
7  import java.net.*;
8  import java.util.*;
9
10 import com.deitel.messenger.*;
11 import com.deitel.messenger.sockets.*;
12
13 public class SubprocesoReceptorPaquetes extends Thread {
14
15    // objeto EscuchaMensajes al que deben entregarse los mensajes
16    private EscuchaMensajes escuchaMensajes;
17
18    // objeto MulticastSocket para recibir mensajes de transmisión
19    private MulticastSocket socketMulticast;
20
21    // objeto InetAddress del grupo para los mensajes
22    private InetAddress grupoMulticast;
23
24    // bandera para terminar objeto SubprocesoReceptorPaquetes
25    private boolean seguirEscuchando = true;
26
27    // constructor de SubprocesoReceptorPaquetes
28    public SubprocesoReceptorPaquetes( EscuchaMensajes escucha )
29    {
30       // invocar al constructor de la superclase para nombrar el objeto Thread
31       super( "SubprocesoReceptorPaquetes" );
32
33       // establecer objeto EscuchaMensajes
34       escuchaMensajes = escucha;
35
36       // conectarse con objeto MulticastSocket para realizar transmisión múltiple
            de dirección y puerto
37       try {
38          socketMulticast = new MulticastSocket(
39             ConstantesSocketMessenger.PUERTO_ESCUCHA_MULTICAST );
40
41          grupoMulticast = InetAddress.getByName(
42             ConstantesSocketMessenger.DIRECCION_MULTICAST );
43
44          // unirse al grupo de transmisión múltiple para recibir los mensajes
45          socketMulticast.joinGroup( grupoMulticast );
46
47          // establecer tiempo fuera de 5 segundos al esperar nuevos paquetes
48          socketMulticast.setSoTimeout( 5000 );
49       }
50
51       // manejar posible excepción al conectarse con dirección de transmisión múltiple
52       catch ( IOException excepcionES ) {
53          excepcionES.printStackTrace();
54       }
55
56    } // fin del constructor de SubprocesoReceptorPaquetes
57
58    // escuchar mensajes del grupo de transmisión múltiple
```

Figura 18.19 La clase **SubprocesoReceptorPaquetes** para escuchar nuevos mensajes de transmisión múltiple del **ServidorDeitelMessenger** en un subproceso separado. (Parte 2 de 4.)

```
59    public void run()
60    {
61       // escuchar mensajes hasta ser detenido
62       while ( seguirEscuchando ) {
63
64          // crear búfer para mensaje entrante
65          byte[] bufer =
66             new byte[ ConstantesSocketMessenger.TAMANIO_MENSAJE ];
67
68          // crear objeto DatagramPacket para mensaje entrante
69          DatagramPacket paquete = new DatagramPacket( bufer,
70             ConstantesSocketMessenger.TAMANIO_MENSAJE );
71
72          // recibir nuevo objeto DatagramPacket (llamada de bloqueo)
73          try {
74             socketMulticast.receive( paquete );
75          }
76
77          // manejar posible excepción al agotarse el tiempo para recibir
78          catch ( InterruptedIOException excepcionESInterrupcion ) {
79
80             // continuar con la siguiente iteración para seguir escuchando
81             continue;
82          }
83
84          // manejar excepción al leer el paquete del grupo de transmisión múltiple
85          catch ( IOException excepcionES ) {
86             excepcionES.printStackTrace();
87             break;
88          }
89
90          // colocar datos del mensaje en un objeto String
91          String mensaje = new String( paquete.getData() );
92
93          // eliminar espacio en blanco adicional de la parte final del mensaje
94          mensaje = mensaje.trim();
95
96          // dividir mensaje en tokens para recuperar nombre de usuario y cuerpo del mensaje
97          StringTokenizer tokenizer = new StringTokenizer(
98             mensaje, ConstantesSocketMessenger.SEPARADOR_MENSAJE );
99
100         // ignorar mensajes que no contengan un nombre
101         // de usuario y cuerpo del mensaje
102         if ( tokenizer.countTokens() == 2 )
103
104            // enviar mensaje a objeto EscuchaMensajes
105            escuchaMensajes.mensajeRecibido(
106               tokenizer.nextToken(),    // nombre de usuario
107               tokenizer.nextToken() );  // cuerpo del mensaje
108
109      } // fin de instrucción while
110
111      // dejar grupo de transmisión múltiple y cerrar objeto MulticastSocket
```

Figura 18.19 La clase **SubprocesoReceptorPaquetes** para escuchar nuevos mensajes de transmisión múltiple del **ServidorDeitelMessenger** en un subproceso separado. (Parte 3 de 4.)


```
112      try {
113         socketMulticast.leaveGroup( grupoMulticast );
114         socketMulticast.close();
115      }
116
117      // manejar posible excepción al leer el paquete del grupo de transmisión múltiple
118      catch ( IOException excepcionES ) {
119         excepcionES.printStackTrace();
120      }
121
122   } // fin del método run
123
124   // dejar de escuchar nuevos mensajes
125   public void dejarDeEscuchar()
126   {
127      seguirEscuchando = false;
128   }
129
130 } // fin de la clase SubprocesoReceptorPaquetes
```

Figura 18.19 La clase **SubprocesoReceptorPaquetes** para escuchar nuevos mensajes de transmisión múltiple del **ServidorDeitelMessenger** en un subproceso separado. (Parte 4 de 4.)

Si el programa invoca al método **dejarDeEscuchar** (líneas 125 a 128), termina el ciclo **while** en el método **run** (líneas 62 a 109). En la línea 113 se invoca el método **leaveGroup** de **MulticastSocket** para dejar de recibir mensajes de la dirección de transmisión múltiple. En la línea 114 se invoca el método **close** de **MulticastSocket** para cerrar el objeto **MulticastSocket**. Cuando el método **run** completa su ejecución, termina el objeto **SubprocesoReceptorPaquetes**.

La clase **GUICliente** (figura 18.20) extiende la clase **JFrame** para crear una GUI en la que un usuario pueda enviar y recibir mensajes de conversación. La GUI consiste de un objeto **JTextArea** para mostrar los mensajes entrantes (línea 19), un objeto **JTextArea** para introducir nuevos mensajes (línea 20), objetos **JButton** y **JMenuItem** para conectarse y desconectarse del servidor (líneas 23 a 26) y un objeto **JButton** para enviar mensajes (línea 29). La GUI también contiene un objeto **JLabel** que muestra si el cliente está conectado o desconectado (línea 32).

```
1  // Fig. 18.20: GUICliente.java
2  // GUICliente proporciona una interfaz de usuario para enviar y recibir
3  // mensajes hacia y desde el ServidorDeitelMessenger.
4  package com.deitel.messenger;
5
6  import java.io.*;
7  import java.net.*;
8  import java.awt.*;
9  import java.awt.event.*;
10 import javax.swing.*;
11 import javax.swing.border.*;
12
13 public class GUICliente extends JFrame {
14
15    // objeto JMenu para conectarse al/desconectarse del servidor
16    private JMenu menuServidor;
```

Figura 18.20 **GUICliente** es una subclase de **JFrame** para presentar una GUI en la que se puedan ver y enviar mensajes de conversación. (Parte 1 de 6.)

http://librosysolucionarios.net

```
17
18    // objetos JTextArea para mostrar e introducir mensajes
19    private JTextArea areaMensajes;
20    private JTextArea areaIntroducir;
21
22    // objetos JButton y JMenuItem para conectarse y desconectarse
23    private JButton botonConectar;
24    private JMenuItem elementoMenuConectar;
25    private JButton botonDesconectar;
26    private JMenuItem elementoMenuDesconectar;
27
28    // objeto JButton para enviar mensajes
29    private JButton botonEnviar;
30
31    // objeto JLabel para mostrar el estado de la conexión
32    private JLabel barraEstado;
33
34    // nombreUsuario que se va a agregar a los mensajes salientes
35    private String nombreUsuario;
36
37    // objeto AdministradorMensajes para comunicarse con el servidor
38    private AdministradorMensajes administradorMensajes;
39
40    // objeto EscuchaMensajes para recibir los mensajes entrantes
41    private EscuchaMensajes escuchaMensajes;
42
43    // constructor de GUICliente
44    public GUICliente(AdministradorMensajes administrador)
45    {
46       super( "Deitel Messenger" );
47
48       // establecer el objeto AdministradorMensajes
49       administradorMensajes = administrador;
50
51       // crear objeto MiEscuchaMensajes para recibir mensajes
52       escuchaMensajes = new MiEscuchaMensajes();
53
54       // crear objeto JMenu Servidor
55       menuServidor = new JMenu ( "Servidor" );
56       menuServidor.setMnemonic( 'S' );
57       JMenuBar barraMenus = new JMenuBar();
58       barraMenus.add( menuServidor );
59       setJMenuBar( barraMenus );
60
61       // crear objeto ImageIcon para botones conectar
62       Icon iconoConectar = new ImageIcon(
63          getClass().getResource( "imagenes/Conectar.gif" ) );
64
65       // crear botonConectar y elementoMenuConectar
66       botonConectar = new JButton( "Conectar", iconoConectar );
67       elementoMenuConectar = new JMenuItem( "Conectar", iconoConectar );
68       elementoMenuConectar.setMnemonic( 'C' );
69
70       // crear objeto EscuchaConectar para botones conectar
```

Figura 18.20 `GUICliente` es una subclase de `JFrame` para presentar una GUI en la que se puedan ver y enviar mensajes de conversación. (Parte 2 de 6.)

http://librosysoluciones.net


```
71       ActionListener escuchaConectar = new EscuchaConectar();
72       botonConectar.addActionListener( escuchaConectar );
73       elementoMenuConectar.addActionListener( escuchaConectar );
74
75       // crear objeto ImageIcon para botones desconectar
76       Icon iconoDesconectar = new ImageIcon(
77          getClass().getResource( "imagenes/Desconectar.gif" ) );
78
79       // crear botonDesconectar y elementoMenuDesconectar
80       botonDesconectar = new JButton( "Desconectar", iconoDesconectar );
81       elementoMenuDesconectar = new JMenuItem( "Desconectar", iconoDesconectar );
82       elementoMenuDesconectar.setMnemonic( 'D' );
83
84       // deshabilitar botones desconectar
85       botonDesconectar.setEnabled( false );
86       elementoMenuDesconectar.setEnabled( false );
87
88       // crear objeto EscuchaDesconectar para botones desconectar
89       ActionListener escuchaDesconectar = new EscuchaDesconectar();
90       botonDesconectar.addActionListener( escuchaDesconectar );
91       elementoMenuDesconectar.addActionListener( escuchaDesconectar );
92
93       // agregar objetos JMenuItem para conectar y desconectar a objeto menuServidor
94       menuServidor.add( elementoMenuConectar );
95       menuServidor.add( elementoMenuDesconectar );
96
97       // agregar objetos JButton para conectar y desconectar a panelBotones
98       JPanel panelBotones = new JPanel();
99       panelBotones.add( botonConectar );
100      panelBotones.add( botonDesconectar );
101
102      // crear objeto JTextArea para mostrar mensajes
103      areaMensajes = new JTextArea();
104
105      // deshabilitar edición y habilitar envoltura de palabras al final de la línea
106      areaMensajes.setEditable( false );
107      areaMensajes.setWrapStyleWord( true );
108      areaMensajes.setLineWrap( true );
109
110      // colocar areaMensajes en objeto JScrollPane para habilitar capacidad
            de desplazamiento
111      JPanel panelMensajes = new JPanel();
112      panelMensajes.setLayout( new BorderLayout( 10, 10 ) );
113      panelMensajes.add( new JScrollPane( areaMensajes ),
114          BorderLayout.CENTER );
115
116      // crear objeto JTextArea para introducir nuevos mensajes
117      areaIntroducir = new JTextArea( 4, 20 );
118      areaIntroducir.setWrapStyleWord( true );
119      areaIntroducir.setLineWrap( true );
120      areaIntroducir.setEditable( false );
121
122      // crear objeto Icon para botonEnviar
123      Icon iconoEnviar = new ImageIcon(
124         getClass().getResource( "imagenes/Enviar.gif" ) );
```

Figura 18.20 `GUICliente` es una subclase de `JFrame` para presentar una GUI en la que se puedan ver y enviar mensajes de conversación. (Parte 3 de 6.)

```
125
126         // crear botonEnviar y deshabilitarlo
127         botonEnviar = new JButton( "Enviar", iconoEnviar );
128         botonEnviar.setEnabled( false );
129         botonEnviar.addActionListener(
130
131            new ActionListener() {
132
133               // enviar nuevo mensaje cuando el usuario active botonEnviar
134               public void actionPerformed( ActionEvent evento )
135               {
136                  administradorMensajes.enviarMensaje( nombreUsuario,
137                     areaIntroducir.getText() );
138
139                  // borrar areaIntroducir
140                  areaIntroducir.setText("");
141               }
142            }
143         );
144
145         // distribuir areaIntroducir y botonEnviar en objeto BoxLayout y
146         // agregar objeto Box a panelMensajes
147         Box cuadro = new Box( BoxLayout.X_AXIS );
148         cuadro.add( new JScrollPane( areaIntroducir ) );
149         cuadro.add( botonEnviar );
150         panelMensajes.add( cuadro, BorderLayout.SOUTH );
151
152         // crear objeto JLabel para barraEstado con un borde empotrado
153         barraEstado = new JLabel( "Sin conexión" );
154         barraEstado.setBorder( new BevelBorder( BevelBorder.LOWERED ) );
155
156         // distribuir componentes en objeto JFrame
157         Container contenedor = getContentPane();
158         contenedor.add( panelBotones, BorderLayout.NORTH );
159         contenedor.add( panelMensajes, BorderLayout.CENTER );
160         contenedor.add( barraEstado, BorderLayout.SOUTH );
161
162         // agregar objeto WindowListener para desconectarse cuando el usuario salga
163         addWindowListener (
164
165            new WindowAdapter () {
166
167               // desconectarse del servidor y salir de la aplicación
168               public void windowClosing ( WindowEvent evento )
169               {
170                  administradorMensajes.desconectar( escuchaMensajes );
171                  System.exit( 0 );
172               }
173            }
174         );
175
176      } // fin del constructor de GUICliente
177
178      // objeto EscuchaConectar que escucha las peticiones de los usuarios para
            conectarse al servidor
179      private class EscuchaConectar implements ActionListener {
180
```

Figura 18.20 **GUICliente** es una subclase de **JFrame** para presentar una GUI en la que se puedan ver y enviar mensajes de conversación. (Parte 4 de 6.)


```
181         // conectarse al servidor y habilitar/deshabilitar los componentes de la GUI
182         public void actionPerformed( ActionEvent evento )
183         {
184            // conectarse con el servidor y enrutar mensajes hacia escuchaMensajes
185            administradorMensajes.conectar( escuchaMensajes );
186
187            // pedir el nombre del usuario
188            nombreUsuario = JOptionPane.showInputDialog(
189               GUICliente.this, "Escriba el nombre de usuario:" );
190
191            // borrar areaMensajes
192            areaMensajes.setText( "" );
193
194            // actualizar componentes de la GUI
195            botonConectar.setEnabled( false );
196            elementoMenuConectar.setEnabled( false );
197            botonDesconectar.setEnabled( true );
198            elementoMenuDesconectar.setEnabled( true );
199            botonEnviar.setEnabled( true );
200            areaIntroducir.setEditable( true );
201            areaIntroducir.requestFocus();
202            barraEstado.setText( "Conectado: " + nombreUsuario );
203         }
204
205      } // fin de la clase interna EscuchaConectar
206
207      // EscuchaDesconectar escucha las peticiones de los usuarios para
            desconectarse
208      // del objeto ServidorDeitelMessenger
209      private class EscuchaDesconectar implements ActionListener {
210
211         // desconectarse del servidor y habilitar/deshabilitar los
               componentes de la GUI
212         public void actionPerformed( ActionEvent evento )
213         {
214            // desconectarse del servidor y dejar de enrutar los mensajes
215            // hacia escuchaMensajes
216            administradorMensajes.desconectar( escuchaMensajes );
217
218            // actualizar componentes de la GUI
219            botonEnviar.setEnabled( false );
220            botonDesconectar.setEnabled( false );
221            elementoMenuDesconectar.setEnabled( false );
222            areaIntroducir.setEditable( false );
223            botonConectar.setEnabled( true );
224            elementoMenuConectar.setEnabled( true );
225            barraEstado.setText( "Sin conexión" );
226         }
227
228      } // fin de la clase interna EscuchaDesconectar
229
230      // MiEscuchaMensajes escucha los nuevos mensajes del objeto
            AdministradorMensajes y
231      // muestra los mensajes en areaMensajes, utilizando MostradorMensajes.
232      private class MiEscuchaMensajes implements EscuchaMensajes {
233
```

Figura 18.20 **GUICliente** es una subclase de **JFrame** para presentar una GUI en la que se puedan ver y enviar mensajes de conversación. (Parte 5 de 6.)

http://librosysolucionarios.net

```
234         // al recibirlos, mostrar nuevos mensajes en areaMensajes
235         public void mensajeRecibido( String de, String mensaje )
236         {
237            // anexar mensaje utilizando objeto MostradorMensajes e
238            // invokeLater, asegurando un acceso a areaMensajes seguro para
               los subprocesos
239            SwingUtilities.invokeLater(
240               new MostradorMensajes( de, mensaje ) );
241         }
242      }
243
244      // MostradorMensajes muestra un nuevo mensaje, anexando el mensaje al
245      // objeto JTextArea llamado areaMensajes. Este objeto Runnable debe ejecutarse
246      // solamente en el subproceso Event, ya que modifica a un componente de Swing activo
247      private class MostradorMensajes implements Runnable {
248         private String deUsuario;
249         private String cuerpoMensaje;
250
251         // constructor de MostradorMensajes
252         public MostradorMensajes( String de, String cuerpo )
253         {
254            deUsuario = de;
255            cuerpoMensaje = cuerpo;
256         }
257
258         // mostrar nuevo mensaje en areaMensajes
259         public void run()
260         {
261            // anexar nuevo mensaje
262            areaMensajes.append( "\n" + deUsuario + "> " + cuerpoMensaje );
263
264            // desplazar signo de intercalación al final de areaMensajes para
265            // asegurar que el nuevo mensaje pueda verse en la pantalla
266            areaMensajes.setCaretPosition( areaMensajes.getText().length() );
267         }
268
269      } // fin de la clase interna MostradorMensajes
270
271   } // fin de la clase GUICliente
```

Figura 18.20 `GUICliente` es una subclase de `JFrame` para presentar una GUI en la que se puedan ver y enviar mensajes de conversación. (Parte 6 de 6.)

`GUICliente` utiliza un objeto `AdministradorMensajes` (línea 38) para encargarse de toda la comunicación con el servidor de conversación. Recuerde que `AdministradorMensajes` es una interfaz que permite a `GUICliente` utilizar cualquier implementación de `AdministradorMensajes`. La clase `GUICliente` también utiliza un objeto `EscuchaMensajes` (línea 41) para recibir mensajes entrantes del objeto `AdministradorMensajes`.

El constructor de `GUICliente` (líneas 44 a 176) toma como argumento el objeto `AdministradorMensajes` para comunicarse con el `ServidorDeitelMessenger`. En la línea 49 se establece el objeto `AdministradorMensajes` del objeto `GUICliente`. En la línea 52 se crea una instancia de la clase `MiEscuchaMensajes`, la cual implementa a la interfaz `EscuchaMensajes`. En las líneas 55 a 59 se crea un menú **Servidor** que contiene elementos `JMenuItem` para conectarse y desconectarse del servidor de conversación. En las líneas 62 y 63 se crea un objeto `ImageIcon` para `botonConectar` y `elementoMenuConectar`.

http://librosysolucionarios.net

En la línea 63 se invoca el método `getClass` (heredado de la clase `Object`) para recuperar el objeto `Class` que representa la declaración de la clase `GUICliente`. Después, en la línea 63 se invoca el método *`getResource`* de la clase `Class` para cargar la imagen del botón para conectarse. La máquina virtual de Java carga las declaraciones de las clases en memoria, utilizando un *cargador de clases*. El método `getResource` utiliza el cargador de clases del objeto `Class` para especificar la ubicación de un recurso, como un archivo de imagen. Al especificar las ubicaciones de recursos de esta forma los programas pueden evitar las rutas absolutas o codificadas en el programa, lo cual puede hacer que los programas sean más difíciles de implementar en varios equipos. Al utilizar las técnicas aquí descritas, un applet o aplicación puede cargar archivos desde ubicaciones que sean relativas a la ubicación del archivo `.class` para una clase dada.

En las líneas 66 y 67 se crean `botonConectar` y `elementoMenuConectar`, cada uno con la etiqueta `"Conectar"` y el objeto `Icon` `iconoConectar`. En la línea 68 se invoca el método `setMnemonic` para establecer el carácter nemónico para el acceso a `elementoMenuConectar` mediante el teclado. En la línea 71 se crea una instancia de la clase interna `EscuchaConectar` (declarada en las líneas 179 a 205), la cual implementa a la interfaz `ActionListener` para manejar eventos `ActionEvent` provenientes de `botonConectar` y `elementoMenuConectar`. En las líneas 72 y 73 se agrega `escuchaConectar` como un objeto `ActionListener` para `botonConectar` y `elementoMenuConectar`.

En las líneas 76 y 77 se crea un objeto `ImageIcon` para los componentes `botonDesconectar` y `elementoMenuDesconectar`. En las líneas 80 y 81 se crean `botonDesconectar` y `elementoMenuDesconectar`, cada uno con la etiqueta `"Desconectar"` y el objeto `Icon` `iconoDesconectar`. En la línea 82 se invoca el método `setMnemonic` para permitir el acceso a `elementoMenuDesconectar` mediante el teclado. En las líneas 85 y 86 se invoca el método `setEnabled` con un argumento `false` sobre `botonDesconectar` y `elementoMenuDesconectar` para deshabilitar esos componentes. Esto evita que el usuario trate de desconectarse del servidor, porque el cliente no está todavía conectado. En la línea 89 se crea una instancia de la clase interna `EscuchaDesconectar` (declarada en las líneas 209 a 228), la cual implementa a la interfaz `ActionListener` para manejar eventos `ActionEvent` provenientes de `botonDesconectar` y `elementoMenuDesconectar`. En las líneas 90 y 91 se agrega `escuchaDesconectar` como un objeto `ActionListener` para `botonDesconectar` y `elementoMenuDesconectar`.

En las líneas 94 y 95 se agregan `elementoMenuConectar` y `elementoMenuDesconectar` al menú **Servidor**. En las líneas 98 a 100 se crea un objeto `JPanel` y se agregan `botonConectar` y `botonDesconectar` a ese objeto `JPanel`. En la línea 103 se crea el área de texto `areaMensajes`, en la cual el cliente muestra los mensajes entrantes. En la línea 106 se invoca el método `setEnabled` con un argumento `false`, para deshabilitar `areaMensajes`. En las líneas 107 y 108 se invocan los métodos `setWrapStyleWord` y `setLineWrap` de `JTextArea` para habilitar la envoltura de palabras en `areaMensajes`. Si un mensaje es más largo que la anchura de `areaMensajes`, este componente envolverá el texto después de la última palabra que pueda acomodarse en cada línea, facilitando la lectura de los mensajes más largos. En las líneas 113 y 114 se crea un objeto `JPanel` para el componente `areaMensajes` y se agrega `areaMensajes` al objeto `JPanel` en un objeto `JScrollPane`.

En la línea 117 se crea el objeto `JTextArea` llamado `areaIntroducir` para escribir nuevos mensajes. En las líneas 118 y 119 se habilita la envoltura de palabras y líneas, y en la línea 120 se deshabilita la capacidad de editar el componente `areaIntroducir`. Cuando el cliente se conecta al servidor de conversación, `EscuchaConectar` habilita al componente `areaIntroducir` para permitir al usuario que escriba nuevos mensajes.

En las líneas 123 y 124 se crea un objeto `ImageIcon` para `botonEnviar`. En la línea 127 se crea `botonEnviar`, en el cual el usuario puede hacer clic para enviar un mensaje. En la línea 128 se deshabilita `botonEnviar`; el objeto `EscuchaConectar` habilita el `botonEnviar` cuando el cliente se conecta con el servidor de conversación. En las líneas 129 a 143 se agrega un objeto `ActionListened` a `botonEnviar`. En las líneas 136 y 137 se invoca el método `enviarMensaje` de la interfaz `AdministradorMensajes` con el texto de `nombreUsuario` y `areaIntroducir` como argumentos. Esta instrucción envía el nombre de usuario y el mensaje como un nuevo mensaje de conversación a `ServidorDeitelMessenger`. En la línea 140 se borra el componente `areaIntroducir` para el siguiente mensaje.

En las líneas 147 a 150 se utiliza un contenedor `Box` horizontal para ordenar los componentes `areaIntroducir` y `botonEnviar`. En la línea 148 se coloca `areaIntroducir` en un objeto `JScrollPane`

para habilitar el desplazamiento de los mensajes largos. En la línea 150 se agrega el objeto `Box` que contiene a `areaIntroducir` y `botonEnviar` a la región `SOUTH` de `panelMensajes`. En la línea 153 se crea el objeto `JLabel` llamado `barraEstado`. Esta etiqueta muestra si el cliente está conectado o desconectado del servidor de conversación. En la línea 154 se invoca el método `setBorder` de la clase `JLabel` y se crea un nuevo objeto `BevelBorder` de tipo `BevelBorder.LOWERED`. Este borde hace que la etiqueta aparezca empotrada, como es común en las barras de estado de muchas aplicaciones. En las líneas 158 a 160 se agregan `panelBotones`, `panelMensajes` y `barraEstado` al panel de contenido de `GUICliente`.

En las líneas 163 a 174 se agrega un objeto `WindowListener` a `GUICliente`. En la línea 170 se invoca el método `desconectar` de la interfaz `AdministradorMensajes` para desconectarse del servidor de conversación, en caso de que el usuario desee abandonar la aplicación mientras aún siga conectado. Después, en la línea 171 se termina la aplicación.

La clase interna `EscuchaConectar` (líneas 179 a 205) maneja los eventos provenientes de `botonConectar` y `elementoMenuConectar`. En la línea 185 se invoca el método `conectar` de `AdministradorMensajes` para conectarse con el servidor de conversación. En la línea 185 se pasa el objeto `EscuchaMensajes` como argumento para el método `conectar`, al cual deben entregarse los nuevos mensajes. En las líneas 188 y 189 se pide al usuario un nombre de usuario, y en la línea 98 se borra el `areaMensajes`. En las líneas 195 a 200 se habilitan los componentes para desconectarse del servidor, para enviar mensajes y deshabilitar componentes para conectarse al servidor. En la línea 201 se invoca el método `requestFocus` de `areaIntroducir`, para colocar el cursor de introducción de texto en el componente `areaIntroducir`, de manera que el usuario pueda empezar a escribir un mensaje.

La clase interna `EscuchaDesconectar` (líneas 209 a 228) maneja los eventos provenientes de `botonDesconectar` y `elementoMenuDesconectar`. En la línea 216 se invoca el método `desconectar` de `AdministradorMensajes` para desconectarse del servidor de conversación. En las líneas 219 a 224 se deshabilitan los componentes para enviar mensajes y los componentes para desconectarse, después se habilitan los componentes para conectarse al servidor de conversación.

La clase interna `MiEscuchaMensajes` (líneas 232 a 242) implementa a la interfaz `EscuchaMensajes` para recibir los mensajes entrantes del objeto `AdministradorMensajes`. Cuando se recibe un nuevo mensaje, el objeto `AdministradorMensajes` invoca al método `mensajeRecibido` (líneas 235 a 241) con el nombre de usuario del emisor y el cuerpo del mensaje. En las líneas 239 y 240 se invoca el método `invokeLater` de `SwingUtilities` con un objeto `MostradorMensajes` que anexa el nuevo mensaje al `areaMensajes`. Recuerde que en el capítulo 16 vimos que los componentes de Swing deben utilizarse solamente desde el subproceso despachador de eventos. El método `mensajeRecibido` es invocado por el `SubprocesoReceptorPaquetes` en la clase `AdministradorSocketsMensajes`, y por lo tanto no puede anexar el texto de mensaje directamente al `areaMensajes`, ya que esto ocurriría en `SubprocesoReceptorPaquetes` y no en el subproceso despachador de eventos.

La clase interna `MostradorMensajes` (líneas 247 a 269) implementa a la interfaz `Runnable` para proporcionar una manera de anexar texto al `areaMensajes`, que sea segura para los subprocesos. El constructor de `MostradorMensajes` (líneas 252 a 256) toma como argumentos el nombre de usuario y el mensaje a enviar. El método `run` (líneas 259 a 267) anexa el nombre de usuario, `"> "` y el `cuerpoMensaje` al componente `areaMensajes`. En la línea 266 el texto se desplaza a la parte inferior de `areaMensajes`, para mostrar el mensaje que se acaba de recibir.

La clase `DeitelMessenger` (figura 18.21) inicia el cliente para el `ServidorDeitelMessenger`. En las líneas 15 a 18 se crea un nuevo objeto `AdministradorSocketsMensajes` para conectarse al `ServidorDeitelMessenger` con la dirección IP especificada como argumento de línea de comandos en la aplicación (o localhost, si no se proporciona una dirección). En las líneas 21 a 24 se crea un objeto `GUICliente` para el `AdministradorMensajes`, se establece el tamaño de la `GUICliente` y se hace visible.

Ejecución de la aplicación cliente `DeitelMessenger`

Para ejecutar el cliente `DeitelMessenger`, abra una ventana de comandos y cambie al directorio en donde se encuentra el paquete `com.deitel.messenger.sockets.cliente` (es decir, el directorio en el que se encuentra `com`). Después, escriba:

```
java com.deitel.messenger.sockets.client.DeitelMessenger
```

http://librosysolucionarios.net


```
1  // Fig. 18.21: DeitelMessenger.java
2  // DeitelMessenger es una aplicación de conversación (chat) que utiliza a
   GUICliente
3  // y AdministradorSocketsMensajes para comunicarse con el
   ServidorDeitelMessenger.
4  package com.deitel.messenger.sockets.cliente;
5
6  import com.deitel.messenger.*;
7
8  public class DeitelMessenger {
9
10    public static void main( String args[] )
11    {
12       AdministradorMensajes administradorMensajes;
13
14       // crear nuevo objeto DeitelMessenger
15       if ( args.length == 0 )
16          administradorMensajes = new AdministradorSocketsMensajes( "localhost" );
17       else
18          administradorMensajes = new AdministradorSocketsMensajes( args[ 0 ] );
19
20       // crear la GUI para objeto AdministradorSocketsMensajes
21       GUICliente guiCliente = new GUICliente( administradorMensajes );
22       GUICliente.setSize( 300, 400 );
23       GUICliente.setResizable( false );
24       GUICliente.setVisible( true );
25    }
26
27 } // fin de la clase DeitelMessenger
```

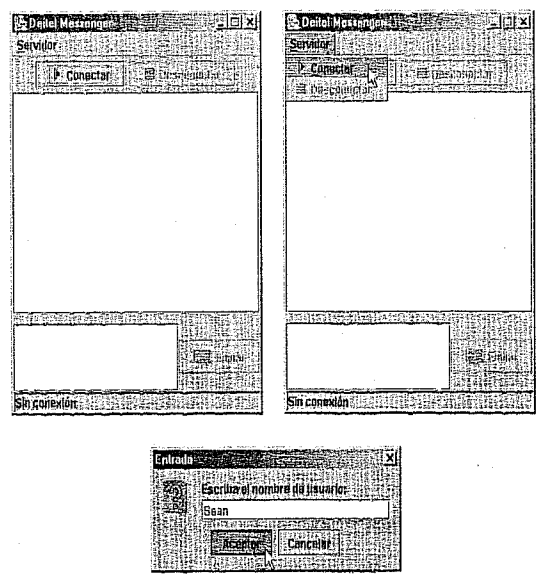


Figura 18.21 La aplicación `DeitelMessenger` para participar en una sesión de conversación del `ServidorDeitelMessenger`. (Parte 1 de 2.)

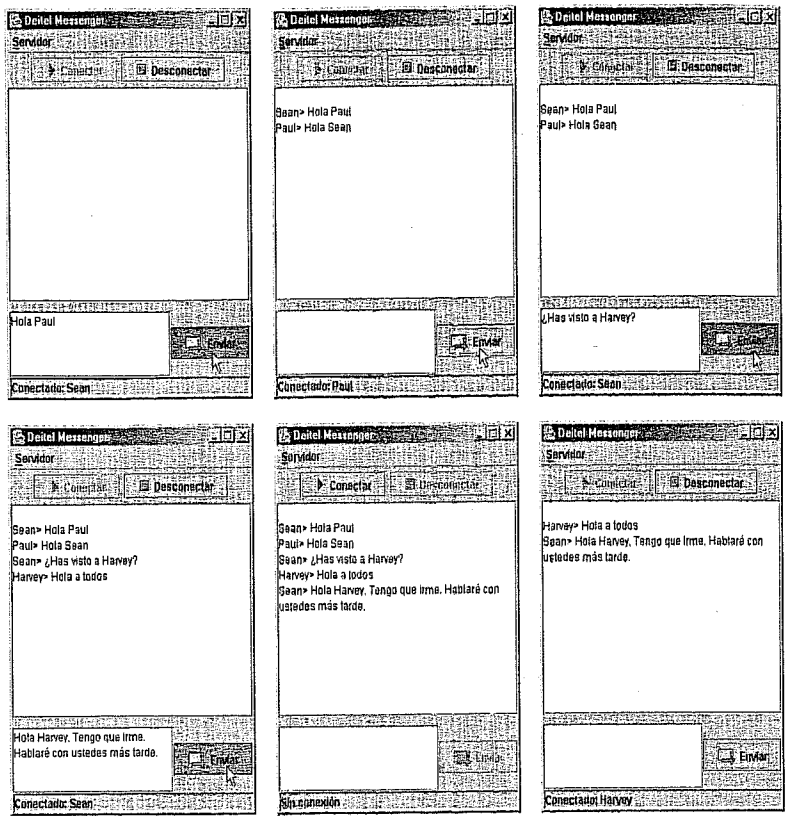


Figura 18.21 La aplicación `DeitelMessenger` para participar en una sesión de conversación del `ServidorDeitelMessenger`. (Parte 2 de 2.)

para ejecutar el cliente y conectarse al `ServidorDeitelMessenger` que se esté ejecutando en su equipo local. Si el servidor reside en otra computadora, utilice el comando anterior agregando el nombre de host o la dirección IP de esa computadora. El comando anterior es equivalente a:

```
java com.deitel.messenger.sockets.cliente.DeitelMessenger localhost
```

o

```
java com.deitel.messenger.sockets.cliente.DeitelMessenger 127.0.0.1
```

Resumen del ejemplo práctico `Deitel Messenger`

El ejemplo práctico del Deitel Messenger es una aplicación considerable que utiliza muchas características en Java de nivel intermedio, como el uso de redes con objetos `Socket`, `DatagramPacket` y `MulticastSocket`, el subprocesamiento múltiple y la GUI de Swing. Este ejemplo práctico también demuestra las buenas prácticas de ingeniería de software al separar la interfaz de la implementación, lo cual permite a los desarrolladores crear objetos `AdministradorMensajes` para los distintos protocolos de red, y objetos `EscuchaMensajes` que proporcionen distintas interfaces de usuario. Ahora usted podrá aplicar estas técnicas en sus propios proyectos más complejos de Java.

http://librosysolucionarios.net

18.11 Generalidades acerca del trabajo en red con NIO

Antes de la J2SE 1.4, todas las llamadas de E/S utilizaban el bloqueo. Cuando se llamaba a una operación de entrada o salida, el programa se detenía hasta que se completaba la operación de E/S, o en el caso de las operaciones de lectura con sockets, hasta que expiraba el tiempo de interrupción configurado opcionalmente. En el caso de los clientes y servidores esto implica muchos problemas. En una aplicación cliente/servidor con un solo subproceso, si un cliente (cliente B) trata de conectarse al servidor mientras que otro cliente (cliente A) se encuentra conectado, el cliente B es bloqueado hasta que el cliente A se desconecta del servidor. La comunicación concurrente entre un servidor y varios clientes requiere de varios subprocesos. Para cada petición de E/S entrante de los clientes, se origina un subproceso servidor separado para dar servicio a la petición de E/S. En aplicaciones con muchos clientes (por ejemplo, en los servidores Web) esto produce una sobrecarga, al mismo tiempo que se reduce la escalabilidad de esos servidores. Las nuevas APIs de E/S en la J2SE 1.4 permiten a los desarrolladores utilizar la *E/S sin bloqueo* en sus aplicaciones. Con la E/S sin bloqueo, cuando se llama a una operación de entrada o salida, esta operación regresa inmediatamente con la información que esté disponible y no se detiene la ejecución del programa. Ahora los servidores pueden interactuar con varias conexiones de clientes en forma simultánea, sin tener que crear un nuevo subproceso para cada conexión.

El intento continuo de leer datos de cada conexión desperdicia tiempo del procesador. La *selección de disponibilidad* es un medio para mejorar la eficiencia del proceso de transferir datos. La selección de disponibilidad es la comprobación simultánea de varios canales, para determinar cuál(es) de ellos está(n) listo(s) para realizar operaciones de E/S. En vez de consultar a cada canal, el programador determina cuáles canales están listos con un solo paso. Cada canal que esté listo se procesa en forma separada, y los que no están listos se ignoran. La selección de disponibilidad se lleva a cabo mediante *selectores*. Un selector supervisa sus canales registrados, identifica los canales que están listos para la E/S y pasa las peticiones de los clientes a los canales correspondientes.

La interfaz `SelectableChannel` (en el paquete `java.nio.channels`) proporciona un marco de trabajo para crear y utilizar canales sin bloqueo. Un objeto `SelectableChannel` es un canal que soporta la selección de disponibilidad, lo cual significa que el canal puede ser consultado para determinar si está listo para realizar operaciones de E/S. Las subclases más útiles de `SelectableChannel` son `ServerSocketChannel` y `SocketChannel`. Los objetos de estas clases se encargan de la comunicación en la red entre un servidor y un cliente, y pueden trabajar en modo sin bloqueo. La clase `Selector` (también en el paquete `java.nio.channels`) proporciona la funcionalidad de los selectores.

Programa de conversación `DeitelMessenger` con E/S sin bloqueo

En la sección 18.10 utilizamos el subprocesamiento múltiple para crear un servidor de conversación `DeitelMessenger` que puede manejar muchos clientes en forma simultánea. Aquí crearemos un servidor con un solo subproceso, sin bloqueo, que utiliza la nueva API de E/S.

`ServidorDeitelMessengerSinBloqueo` (figura 18.22) es el servidor `DeitelMessenger` sin bloqueo. En la línea 22 se declara un objeto `Charset` que se utiliza para codificar y decodificar los mensajes enviados entre el servidor y los clientes. En la línea 23 se declara un objeto `ByteBuffer`, el cual se utiliza para almacenar los bytes escritos en el canal de socket. En la línea 23 se declara e inicializa un objeto `ByteBuffer`, con capacidad de 512 que se utiliza para almacenar los bytes que se leen desde el canal de socket. El constructor (líneas 26 a 60) establece la GUI y cierra los objetos `ServerSocketChannel` y `Selector` al cerrar la ventana. El método `ejecutarServidor` (líneas 63 a 87) especifica un objeto `Charset` para codificar y decodificar mensajes, crea un objeto `ServerSocketChannel` y llama al método `obtenerConexion` (líneas 90 a 154) para esperar la conexión de un cliente. En la línea 69 se invoca el método estático `forName` de la clase `Charset` para obtener un objeto `Charset` para `"UTF-8"`, un formato de codificación de anchura de variable que requiere de uno a cuatro bytes para expresar cada uno de los caracteres Unicode. Vea el apéndice G para obtener más información acerca de Unicode. Un objeto `ServerSocketChannel` realiza la misma tarea que `ServerSocket`: escucha las conexiones de sockets. La principal diferencia entre `ServerSocket` y `ServerSocketChannel` es que `ServerSocketChannel` puede trabajar en modo sin bloqueo. Un objeto `ServerSocketChannel` se crea invocando al método `static open` de la clase `ServerSocketChannel` (línea 72). En las líneas 73 y 74 se enlaza un puerto al objeto `ServerSocket` asociado con el objeto `ServerSocketChannel`. El método `socket` de la clase `ServerSocket-`

`Channel` devuelve el objeto `ServerSocket` asociado con el canal. En la línea 75 se pasa `false` al método `configureBlocking` para hacer que el objeto `ServerSocketChannel` opere en modo sin bloqueo, de manera que varios clientes puedan conectarse al servidor concurrentemente.

```
1  // Fig. 18.22: ServidorDeitelMessengerSinBloqueo.java
2  // Establecer un servidor de conversación sin bloqueo que recibirá una conexión de un
3  // cliente y repetirá el mensaje del cliente a todos los demás clientes conectados.
4  package com.deitel.messenger.sockets.servidor;
5
6  import java.io.*;
7  import java.nio.*;
8  import java.nio.channels.*;
9  import java.nio.channels.spi.*;
10 import java.nio.charset.*;
11 import java.net.*;
12 import java.util.*;
13 import java.awt.event.*;
14 import javax.swing.*;
15
16 public class ServidorDeitelMessengerSinBloqueo extends JFrame {
17    private ServerSocketChannel canalSocketServidor;
18    private Selector selector;
19    private Vector sockets = new Vector();
20    private int contador = 0;
21    private JTextArea areaPantalla;
22    private Charset conjuntoChar;
23    private ByteBuffer bufferEscritura;
24    private ByteBuffer bufferLectura = ByteBuffer.allocate( 512 );
25
26    public ServidorDeitelMessengerSinBloqueo()
27    {
28       super( "Servidor DeitelMessenger" );
29
30       areaPantalla = new JTextArea();
31       getContentPane().add( new JScrollPane( areaPantalla ) );
32
33       setSize( 200, 300 );
34       setVisible( true );
35
36       // cerrar canal de socket y selector del servidor al cerrar la ventana
37       addWindowListener(
38
39          new WindowAdapter() {
40
41             public void windowClosing( WindowEvent eventoVentana )
42             {
43                // cerrar canal de socket y selector del servidor
44                try {
45                   canalSocketServidor.close();
46                   selector.close();
47                }
48                catch( IOException excepcionES ) {
49                   excepcionES.printStackTrace();
50                }
```

Figura 18.22 Servidor de mensajería sin bloqueo. (Parte 1 de 5.)


```
51                finally {
52                   System.exit( 0 );
53                }
54             }
55
56       } // fin de la clase interna WindowAdapter
57
58    ); // fin de addWindowListener
59
60 } // fin del constructor
61
62 // configurar y ejecutar el servidor
63 public void ejecutarServidor()
64 {
65    // configurar servidor para que reciba conexiones; procesar las conexiones
66    try {
67
68       // especificar el conjunto de caracteres usado para codificar/
            decodificar mensajes
69       conjuntoChar = Charset.forName( "UTF-8" );
70
71       // crear un objeto ServerSocketChannel
72       canalSocketServidor = ServerSocketChannel.open();
73       canalSocketServidor.socket().bind(
74          new InetSocketAddress( 12345 ) );
75       canalSocketServidor.configureBlocking( false );
76
77       // esperar una conexión.
78       obtenerConexion();
79
80    } // fin del bloque try
81
82    // procesar posibles problemas con E/S
83    catch ( Exception excepcionES ) {
84       excepcionES.printStackTrace();
85    }
86
87 } // fin del método ejecutarServidor
88
89 // esperar a que llegue una conexión, después mostrar la información
      sobre la conexión
90 private void obtenerConexion() throws Exception
91 {
92    // Selector para peticiones entrantes
93    selector = SelectorProvider.provider().openSelector();
94    canalSocketServidor.register(
95       selector, SelectionKey.OP_ACCEPT, null );
96
97    // procesar peticiones entrantes
98    while ( selector.select() > 0 ) {
99
100      // alistar canales para E/S
101      Set clavesListos = selector.selectedKeys();
102      Iterator iterador = clavesListos.iterator();
103
104      // para cada canal listo, procesar petición
105      while ( iterador.hasNext() ) {
```

Figura 18.22 Servidor de mensajería sin bloqueo. (Parte 2 de 5.)

http://librosysolucionarios.net

```
106         SelectionKey clave = ( SelectionKey )iterador.next();
107         iterador.remove();
108
109            if ( clave.isAcceptable() ) {  // listo para la conexión
110
111               // crear conexión
112               ServerSocketChannel siguienteListo =
113                  ( ServerSocketChannel ) clave.channel();
114               SocketChannel canalSocket = siguienteListo.accept();
115
116               if ( canalSocket != null ) {
117                  canalSocket.configureBlocking( false );
118                  sockets.add( canalSocket.socket() );
119                  contador++;
120
121                  SwingUtilities.invokeLater(
122
123                     new Runnable() {
124
125                        public void run()
126                        {
127                           areaPantalla.append(
128                              "\nConexión con Cliente " + contador );
129                        }
130                     }
131                  );
132
133                  // registrar operación de lectura para canalSocket
134                  SelectionKey claveLectura = canalSocket.register(
135                     selector, SelectionKey.OP_READ, null );
136
137               } // fin de instrucción if canalSocket != null
138
139            } // fin de instrucción if clave.isAcceptable
140
141            else if ( clave.isReadable() ) {  // listo para lectura
142
143               // obtener canalSocket listo para lectura
144               SocketChannel canalSocket =
145                  ( SocketChannel ) clave.channel();
146
147               leerMensaje( canalSocket );
148            }
149
150         } // fin del procesamiento de cada canal
151
152      } // fin del procesamiento de peticiones entrantes
153
154   } // fin del método obtenerConexion
155
156   // enviar mensaje al cliente
157   private void escribirMensaje( String mensaje ) throws IOException
158   {
159      Socket socket;
160      SocketChannel canalSocket;
```

Figura 18.22 Servidor de mensajería sin bloqueo. (Parte 3 de 5.)


```
161
162      // repetir mensaje de vuelta a todos los clientes conectados
163      for ( int i = 0; i < sockets.size(); i++ ) {
164         socket = ( Socket ) sockets.elementAt( i );
165         canalSocket = socket.getChannel();
166
167         // enviar mensaje al cliente
168         try {
169
170            // convertir mensaje en bytes en conjuntoChar
171            buferEscritura = conjuntoChar.encode( mensaje );
172
173            // escribir mensaje en canalSocket
174            canalSocket.write( buferEscritura );
175         }
176
177         // procesar posibles problemas al enviar el objeto
178         catch ( IOException excepcionES ) {
179            excepcionES.printStackTrace();
180            canalSocket.close();
181            sockets.remove( socket );
182         }
183
184      } // fin de instrucción for
185
186   } // fin del método escribirMensaje
187
188   // leer mensaje del cliente
189   private void leerMensaje( SocketChannel canalSocket )
190      throws IOException
191   {
192      // leer mensaje
193      try {
194
195         if ( canalSocket.isOpen() ) {
196            buferLectura.clear();
197            canalSocket.read( buferLectura );
198            buferLectura.flip();
199            CharBuffer mensajeChar = conjuntoChar.decode( buferLectura );
200            String mensaje = mensajeChar.toString().trim();
201
202            // eliminar y cerrar la conexión cuando el cliente se desconecte
203            if ( mensaje.indexOf( "Desconectar" ) >= 0 ) {
204               sockets.remove( canalSocket.socket() );
205               canalSocket.close();
206            }
207            else
208               escribirMensaje( mensaje );
209
210         } // fin de instrucción if
211
212      } // fin del bloque try
213
214      catch ( IOException excepcionES ) {
215         excepcionES.printStackTrace();
```

http://librosysolucionarios.net

Figura 18.22 Servidor de mensajería sin bloqueo. (Parte 4 de 5.)

```
216             sockets.remove( canalSocket.socket() );
217             canalSocket.close();
218          }
219
220       } // fin del método leerMensaje
221
222       public static void main( String args[] )
223       {
224          ServidorDeitelMessengerSinBloqueo aplicacion =
225             new ServidorDeitelMessengerSinBloqueo();
226          aplicacion.ejecutarServidor();
227       }
228
229    } // fin de la clase ServidorDeitelMessengerSinBloqueo
```

Figura 18.22 Servidor de mensajería sin bloqueo. (Parte 5 de 5.)

El método **obtenerConexion** (líneas 90 a 154) espera las conexiones de los clientes y administra los canales que están listos para las operaciones de E/S. En la línea 93 se crea un objeto **Selector** para las peticiones entrantes de los clientes, invocando al método estático **open** de la clase **Selector**. En las líneas 94 y 95 se registra el objeto **canalSocketServidor** con el **selector**. El método **register** toma tres argumentos: un objeto **Selector** con el cual está registrado el canal, un entero que indica el conjunto de operaciones en las que el canal está interesado y un objeto **Object** (es decir, ciertos datos específicos de la aplicación, que pueden ser **null**) que es un adjunto al objeto **SelectionKey**, el cual es devuelto por el método **register**. Un objeto **SelectionKey** contiene dos conjuntos: uno contiene información acerca de cuáles operaciones le interesan al canal registrado, y el otro contiene información acerca de cuáles operaciones está listo para realizar el canal registrado. Un objeto **ServerSocketChannel** estaría interesado en una operación de conexión de socket, por lo que el segundo argumento que se pasa al método **register** es **SelectionKey.OP_ACCEPT**, el cual indica la operación de aceptación de socket. La estructura **while** (líneas 98 a 152) escucha y procesa las peticiones de los clientes. En la línea 98 se invoca el método **select** del objeto **Selector** para seleccionar un conjunto de objetos **SelectionKey**, cuyos canales están listos para operaciones de E/S. En las líneas 101 y 102 se obtiene el conjunto de claves y un iterador del conjunto. El método **selectedKeys** de la clase **Selector** devuelve un objeto **Set** que representa la clave seleccionada por el selector. Todas las claves seleccionadas son de aceptación, lectura o escritura.

La estructura **while** (líneas 105 a 150) procesa las peticiones para cada canal que esté listo para la E/S. En las líneas 106 a 107 se obtiene la siguiente clave en el iterador y se elimina la clave actual, ya que ésta se está manejando. El método **isAcceptable** de la clase **SelectionKey** devuelve **true** cuando el canal asociado a la clave está listo para aceptar una nueva conexión de socket. En las líneas 112 a 137 se crea una conexión de socket **SocketChannel** (líneas 112 a 114) y se registra el canal con el selector para operaciones de lectura (líneas 134 y 135), si el método **isAcceptable** devuelve **true**. En las líneas 108 y 109 se obtiene el objeto **ServerSocketChannel** para el cual se creó la clave, invocando al método **channel** del objeto **SelectionKey** llamado **clave**. En la línea 114 se invoca el método **accept** de **siguienteListo** para crear una conexión de socket. El valor de retorno del método **accept** es un objeto **SocketChannel** que es el canal para la nueva conexión, o **null** si no hay una conexión de cliente pendiente. Si el método **accept** devuelve un objeto **SocketChannel**, en la línea 117 se invoca el método **configureBlocking** con el valor **boolean false** para indicar que el objeto **SocketChannel** recién creado funciona en modo sin bloqueo. En las líneas 118 a 131 se obtiene el objeto **Socket** asociado con el objeto **SocketChannel** invocando al método **socket** de **SocketChannel**, se agrega el socket a un arreglo de sockets y se muestra en pantalla que se realizó una conexión. En las líneas 134 y 135 se registra el objeto **SocketChannel** con el **selector** para una operación de lectura, utilizando **SelectionKey.OP_READ**.

El método **isReadable** de la clase **SelectionKey** devuelve **true** cuando el canal del objeto **SelectionKey** está listo para la operación de lectura. Si el método **isReadable** del objeto **SelectionKey** devuelve **true** en la línea 141, en las líneas 144 y 145 se obtiene el objeto **SocketChannel** asociado con la clave y en la línea 147 se llama al método **leerMensaje** para procesar el mensaje enviado del cliente.

http://librosysolucionarios.net

El método **escribirMensaje** (líneas 157 a 186) envía el mensaje a todos los clientes conectados con el servidor. Para cada socket almacenado en el vector **sockets** (líneas 163 a 184), en las líneas 154 a 165 se obtiene el objeto **SocketChannel** para el socket, en la línea 171 se invoca el método **encode** de la clase **Charset** para convertir el mensaje en un objeto **ByteBuffer** y en la línea 174 se invoca el método **write** de la clase **SocketChannel** para enviar el mensaje (en bytes) al cliente.

El método **readMessage** (líneas 189 a 220) se invoca cada vez que un canal está listo para una operación de lectura. En la línea 195 se comprueba si el canal está abierto, invocando al método **isOpen** de **SocketChannel**. En la línea 196 se invoca el método **clear** de **ByteBuffer** para borrar el búfer, de manera que la siguiente operación de lectura pueda colocar el mensaje empezando en la posición cero. En la línea 197 se invoca el método **read** del objeto **socketChannel** para leer una secuencia de bytes (en este caso, 512 bytes) en el objeto **ByteBuffer** llamado **buferLectura**. Después de la lectura, la posición en cada **buferLectura** se establece en la posición después del último byte leído. En la línea 198 se invoca el método **flip** para establecer el límite en la posición, y la posición en cero, de manera que podamos obtener el mensaje del búfer. En la línea 199 se invoca el método **decode** de **charSet** para obtener un objeto **CharBuffer** que contenga el mensaje. En la línea 200 se invoca el método **toString** en el objeto **CharBuffer** para obtener la representación de cadena. Si el cliente envía una petición de desconexión, en la línea 204 se elimina la conexión del objeto **Vector** llamado **sockets**, y en la línea 205 se cierra la conexión de socket. De no ser así, en la línea 208 se hace una llamada al método **escribirMensaje** para enviar el mensaje a todos los clientes actualmente conectados.

Reutilizamos las clases **GUICliente** (figura 18.20), **EscuchaMensajes** (figura 18.13) y **AdministradorMensajes** (figura 18.16) para crear la aplicación cliente. La clase **AdministradorSocketsMensajes2** (figura 18.23) administra los procesos de conexión, desconexión y la comunicación para el cliente.

```
1  // Fig. 18.23: AdministradorSocketsMensajes2.java
2  // AdministradorSocketsMensajes2 es una clase para objetos capaz de administrar
3  // las comunicaciones con un servidor de mensajes.
4  package com.deitel.messenger.sockets.cliente;
5
6  import java.io.*;
7  import java.nio.*;
8  import java.nio.channels.*;
9  import java.nio.charset.*;
10 import java.net.*;
11 import java.util.*;
12
13 import com.deitel.messenger.*;
14
15 public class AdministradorSocketsMensajes2 implements AdministradorMensajes {
16    private SocketChannel canalSocket;
17    private EscuchaMensajes escuchaMensajes;
18    private String direccionServidor;
19    private SubprocesoReceptor recibirMensaje;
20    private boolean conectado;
21    private Charset conjuntoChar = Charset.forName( "UTF-8" );
22    private ByteBuffer buferEscritura;
23    private ByteBuffer buferLectura = ByteBuffer.allocate( 512 );
24
25    public AdministradorSocketsMensajes2( String host )
```

Figura 18.23 La clase **AdministradorSocketsMensajes2** administra las comunicaciones con el servidor de mensajes. (Parte 1 de 4.)

```
26    {
27       direccionServidor = host;
28       conectado = false;
29    }
30
31    // conectarse con el servidor de mensajes y empezar a recibir el mensaje
32    public void conectar( EscuchaMensajes escucha )
33    {
34       escuchaMensajes = escucha;
35
36       // conectarse con el servidor e iniciar subproceso para recibir mensaje
37       try {
38
39          // crear objeto SocketChannel para realizar conexión con el servidor
40          canalSocket = SocketChannel.open();
41          canalSocket.connect( new InetSocketAddress(
42             InetAddress.getByName( direccionServidor ), 12345 ) );
43
44          // iniciar SubprocesoReceptor para recibir los mensajes enviados
               por el servidor
45          recibirMensaje = new SubprocesoReceptor();
46          recibirMensaje.start();
47
48          conectado = true;
49       }
50       catch ( Exception excepcion ) {
51          excepcion.printStackTrace();
52       }
53    }
54
55    // desconectarse del servidor de mensajes y dejar de recibir mensaje
56    public void desconectar( EscuchaMensajes escucha )
57    {
58       if ( conectado ) {
59
60          // enviar petición de desconexión y dejar de recibir
61          try {
62             enviarMensaje( "", "Desconectar" );
63
64             conectado = false;
65
66             // enviar señal de interrupción al subproceso receptor
67             recibirMensaje.interrupt();
68          }
69          catch ( Exception excepcion ) {
70             excepcion.printStackTrace();
71          }
72       }
73    }
74
75    // enviar mensaje al servidor de mensajes
76    public void enviarMensaje( String nombreUsuario, String cuerpoMensaje )
77    {
78       String mensaje = nombreUsuario + "> " + cuerpoMensaje;
79
```

Figura 18.23 La clase **AdministradorSocketsMensajes2** administra las comunicaciones con el servidor de mensajes. (Parte 2 de 4.)

http://librosysolucionarios.net


```
80       // enviar mensaje al servidor
81       try {
82          bufferEscritura = conjuntoChar.encode( mensaje );
83          canalSocket.write( bufferEscritura );
84       }
85       catch ( IOException excepcionES ) {
86          excepcionES.printStackTrace();
87
88          try {
89             canalSocket.close();
90          }
91          catch ( IOException excepcion ) {
92             excepcion.printStackTrace();
93          }
94       }
95
96    } // fin del método enviarMensaje
97
98    public class SubprocesoReceptor extends Thread
99    {
100      public void run()
101      {
102         int longitudMensaje = 0;
103         String mensaje = "";
104
105         // leer mensajes hasta que el servidor cierre la conexión
106         try {
107
108            // procesar mensajes enviados del servidor
109            do {
110               bufferLectura.clear();
111               canalSocket.read( bufferLectura );
112               bufferLectura.flip();
113               CharBuffer mensajeChar = conjuntoChar.decode( bufferLectura );
114               mensaje = mensajeChar.toString().trim();
115
116               // descomponer mensaje en tokens para recuperar nombre de
                     usuario y cuerpo del mensaje
117               StringTokenizer tokenizer =
118                  new StringTokenizer( mensaje, ">" );
119
120               // ignorar mensajes que no contengan un nombre
121               // de usuario y cuerpo del mensaje
122               if ( tokenizer.countTokens() == 2 )
123
124                  // enviar mensaje a EscuchaMensajes
125                  escuchaMensajes.mensajeRecibido(
126                     tokenizer.nextToken(),    // nombre de usuario
127                     tokenizer.nextToken() ); // cuerpo del mensaje
128
129            } while ( true ); // seguir recibiendo mensajes
130
131         } // fin del bloque try
132
```

Figura 18.23 La clase **AdministradorSocketsMensajes2** administra las comunicaciones con el servidor de mensajes. (Parte 3 de 4.)

```
133          // atrapar posibles problemas al leer del servidor
134          catch ( IOException excepcionES ) {
135             if ( excepcionES instanceof ClosedByInterruptException )
136                System.out.println( "canal de socket cerrado" );
137             else {
138                excepcionES.printStackTrace();
139
140                try {
141                   canalSocket.close();
142                   System.out.println( "canal de socket cerrado" );
143                }
144                catch ( IOException excepcion ) {
145                   excepcion.printStackTrace();
146                }
147             }
148
149          } // fin del bloque catch
150
151       } // fin del método run
152
153    } // fin de la clase interna SubprocesoReceptor
154
155 } // fin de la clase AdministradorSocketsMensajes2
```

Figura 18.23 La clase **AdministradorSocketsMensajes2** administra las comunicaciones con el servidor de mensajes. (Parte 4 de 4.)

El método **conectar** (líneas 32 a 53) establece una conexión con el servidor. En las líneas 40 a 42 se crea una conexión de socket. En la línea 40 se abre un objeto **SocketChannel**, invocando al método **static open** de la clase **SocketChannel**. En las líneas 41 y 42 se conecta el socket de este canal invocando al método **connect** de **SocketChannel** con la dirección **SocketAddress** especificada por **direccionServidor** y el número de puerto **12345**. En las líneas 45 y 46 se crea un objeto **SubprocesoReceptor** para recibir mensajes del servidor e iniciar el subproceso.

El método **enviarMensaje** (líneas 76 a 96) se invoca cuando el usuario hace clic en **Enviar**. En la línea 82 se invoca el método **encode** de **Charset** para codificar el mensaje en bytes. En la línea 83 se invoca el método **write** de **SocketChannel** para enviar el mensaje al servidor en forma de bytes. En caso de que ocurra una excepción **IOException**, que por lo general indica que las futuras operaciones de E/S en ese canal no se llevarán a cabo correctamente, en la línea 89 se invoca el método **close** de **SocketChannel** para cerrar el canal.

La clase interna **SubprocesoReceptor** (líneas 98 a 153) sigue leyendo mensajes del servidor hasta que el subproceso sea interrumpido por el método **desconectar** (líneas 56 a 73), o hasta que el punto remoto termine o restablezca la conexión. En las líneas 110 a 114 se obtiene el mensaje. En las líneas 117 y 118 se crea un objeto **StringTokenizer** para separar el cuerpo del mensaje y el nombre del usuario que envió el mensaje. En la línea 122 se comprueba el número correcto de tokens. En las líneas 125 a 127 se invoca el método **mensajeRecibido** de la interfaz **EscuchaMensajes** para entregar el mensaje entrante al objeto **EscuchaMensajes** del objeto **AdministradorSocketsMensajes2**.

Cuando el usuario hace clic en el botón **Desconectar** o cuando selecciona **Desconectar** en el menú, en las líneas 62 a 67 se envía una petición de desconexión al servidor y se cierra la conexión, si el cliente está conectado. En la línea 62 se envía la petición de desconexión al servidor y en la línea 67 se cierra el objeto **SocketChannel**, interrumpiendo el subproceso receptor. Cuando se interrumpe el subproceso receptor, se cierra el canal que está bloqueando para operaciones de E/S y se lanza una excepción **ClosedByInterruptException**.

La clase **DeitelMessenger2** (figura 18.24) inicia un cliente para el **ServidorDeitelMessengerSinBloqueo**. En las líneas 15 a 18 se crea un nuevo objeto **AdministradorSocketsMensajes2**

http://librosysolucionarios.net

que se conecta al **ServidorDeitelMessengerSinBloqueo**, utilizando la dirección IP especificada como argumento. En las líneas 21 a 24 se crea un objeto **GUICliente** para el **AdministradorMensajes**, se establece el tamaño de la **GUICliente** y se hace visible.

```
1  // Fig. 18.24: DeitelMessenger2.java
2  // DeitelMessenger2 es una aplicación de conversación que utiliza un objeto
3  // GUICliente para comunicarse con el servidor de conversación.
4  package com.deitel.messenger.sockets.cliente;
5
6  import com.deitel.messenger.*;
7
8  public class DeitelMessenger2 {
9
10    public static void main( String args[] )
11    {
12       AdministradorMensajes administradorMensajes;
13
14       // crear nuevo objeto DeitelMessenger
15       if ( args.length == 0 )
16          administradorMensajes = new AdministradorSocketsMensajes2( "localhost" );
17       else
18          administradorMensajes = new AdministradorSocketsMensajes2( args[ 0 ] );
19
20       // crear la GUI para AdministradorSocketsMensajes
21       GUICliente guiCliente = new GUICliente( administradorMensajes );
22       GUICliente.setSize( 300, 400 );
23       GUICliente.setResizable( false );
24       GUICliente.setVisible( true );
25    }
26
27 } // fin de la clase DeitelMessenger2
```

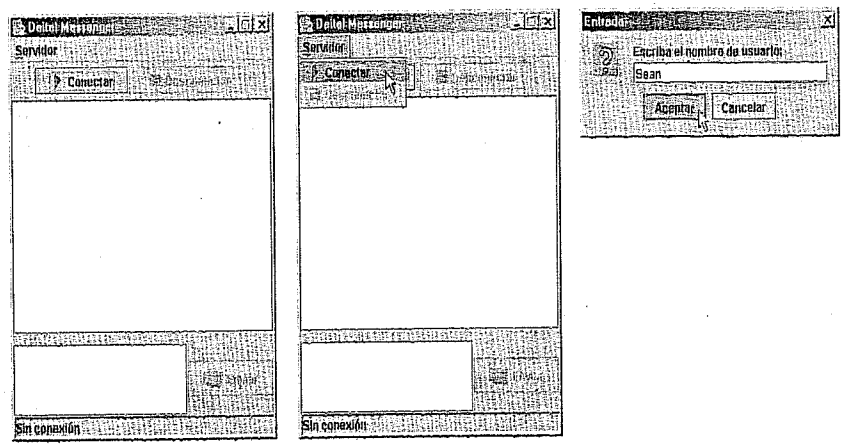


Figura 18.24 La aplicación **DeitelMessenger2** para participar en una sesión de conversación del **ServidorDeitelMessengerSinBloqueo**. (Parte 1 de 2.)

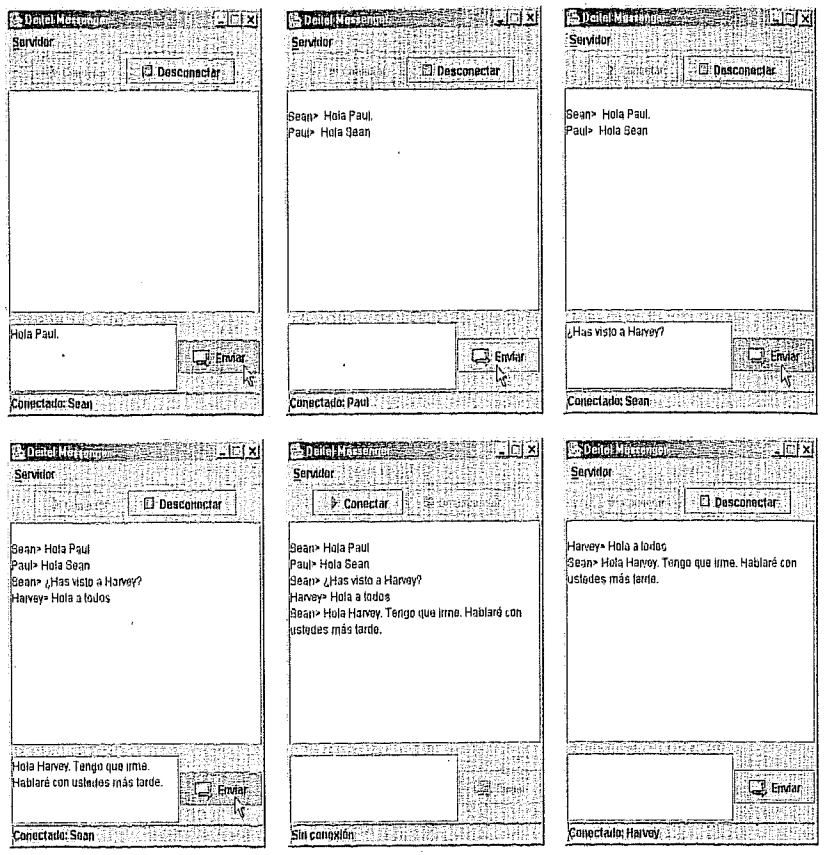


Figura 18.24 La aplicación `DeitelMessenger2` para participar en una sesión de conversación del `ServidorDeitelMessengerSinBloqueo`. (Parte 2 de 2.)

18.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en los paquetes `java.io` y `java.net`

En esta sección le presentaremos los patrones de diseño asociados con el archivo de Java, los flujos y los paquetes de red.

18.12.1 Patrones de diseño de creación

Ahora continuaremos con nuestra discusión sobre los patrones de diseño de creación.

Fábrica abstracta

Al igual que el patrón de diseño Método de fábrica, el *patrón de diseño Fábrica abstracta* permite a un sistema determinar la subclase a partir de la cual se va a instanciar un objeto en tiempo de ejecución. A menudo, esta subclase es desconocida durante el desarrollo. Sin embargo, el patrón de diseño Fábrica abstracta utiliza un objeto conocido como *fábrica*, que utiliza una interfaz para instanciar objetos. Una fábrica devuelve un objeto; en este caso, ese producto es un objeto de una subclases que se determina en tiempo de ejecución.

http://librosysolucionarios.net

La biblioteca de sockets de Java (en el paquete `java.net`) utiliza el patrón de diseño Fábrica abstracta. Un socket describe una conexión, o un flujo de datos, entre dos procesos. La clase `Socket` hace referencia a un objeto de una subclase `SocketImpl` (sección 18.5). La clase `Socket` también contiene una referencia `static` a un objeto que implementa a la interfaz `SocketImplFactory`. El constructor de `Socket` invoca al método `createSocketImpl` de la interfaz `SocketImplFactory` para crear el objeto `SocketImpl`. El objeto que implementa a la interfaz `SocketImplFactory` es la fábrica, y un objeto de una subclase `SocketImpl` es el producto de esa fábrica. El sistema no puede especificar la subclase `SocketImpl` a partir de la cual va a instanciar sino hasta el tiempo de ejecución, ya que el sistema no tiene conocimiento acerca del tipo de implementación de `Socket` que se requiere (por ejemplo, un socket configurado en base a los requerimientos de seguridad de la red local). El método `createSocketImpl` decide de cuál subclase de `SocketImpl` se va a instanciar el objeto en tiempo de ejecución.

18.12.2 Patrones de diseño de estructura

Esta sección concluye nuestra discusión sobre los patrones de diseño de estructura.

Decorador

Examinaremos nuevamente la clase `CrearArchivoSecuencial` (figura 17.7). En las líneas 114 y 115 de esta clase se permite que un objeto `FileOutputStream`, el cual escribe bytes en un archivo, obtenga la funcionalidad de un objeto `ObjectOutputStream`, el cual proporciona métodos para escribir objetos completos en un flujo `OutputStream`. La clase `CrearArchivoSecuencial` parece "envolver" a un objeto `ObjectOutputStream` alrededor de un objeto `FileOutputStream`. El hecho de que podemos agregar en forma dinámica el comportamiento de un objeto `ObjectOutputStream` a un objeto `FileOutputStream` nos evita la necesidad de tener una clase separada llamada `ObjectFileOutputStream`, que implementaría los comportamientos de ambas clases.

En las líneas 127 y 128 de la clase `CrearArchivoSecuencial` se muestra un ejemplo del *patrón de diseño Decorador*, el cual permite a un objeto obtener una funcionalidad adicional en forma dinámica. Mediante el uso de este patrón, los diseñadores no tienen que crear clases separadas innecesarias para añadir responsabilidades a los objetos de una clase dada.

Consideremos un ejemplo más complejo para descubrir cómo el patrón de diseño Decorador puede simplificar la estructura de un sistema. Suponga que queremos mejorar el rendimiento de las operaciones de E/S del ejemplo anterior, utilizando un objeto `BufferedOutputStream`. Utilizando el patrón de diseño Decorador, escribiríamos lo siguiente:

```
salida = new ObjectOutputStream(
   new BufferedOutputStream(
      new FileOutputStream( nombreArchivo ) ) );
```

Podemos combinar objetos de esta manera ya que `ObjectOutputStream`, `BufferedOutputStream` y `FileOutputStream` extienden a la superclase abstracta `OutputStream`, y el constructor de cada subclase toma un objeto `OutputStream` como parámetro. Si los objetos flujo en el paquete `java.io` no utilizaran el patrón Decorador (es decir, si no cumplieran con estos dos requerimientos), el paquete `java.io` tendría que proporcionar las clases `BufferedFileOutputStream`, `ObjectBufferedOutputStream`, `ObjectBufferedFileOutputStream` y `ObjectFileOutputStream`. Considere cuántas clases tendríamos que crear si combináramos aún más objetos flujo sin aplicar el patrón Decorador.

Fachada

Al conducir un auto, usted sabe que si oprime el pedal del acelerador su auto aumenta la velocidad, pero no sabe exactamente por qué el pedal del acelerador hace que su auto acelere. Este principio es la base del *patrón de diseño Fachada*, el cual permite a un objeto (al que se le conoce como *objeto fachada*) proporcionar una interfaz simple para los comportamientos de un *subsistema* (un agregado de objetos que conforman colectivamente una responsabilidad importante del sistema). Por ejemplo, el pedal del acelerador es el objeto fachada para el subsistema de aceleración del auto, el volante de dirección es el objeto fachada para el subsistema de dirección del auto y el freno es el objeto fachada para el subsistema de desaceleración del auto. Un *objeto cliente* utiliza el objeto fachada para acceder a los objetos que están detrás de la fachada. El cliente permanece

sin saber cómo los objetos detrás de la fachada cumplen con sus responsabilidades, por lo que la complejidad del sistema está oculta para el cliente. Cuando usted oprime el pedal del acelerador, actúa como un objeto cliente. El patrón de diseño Fachada reduce la complejidad del sistema, ya que un cliente interactúa solamente con un objeto (la fachada) para acceder a los comportamientos del subsistema que está representado por esa fachada. Este patrón protege a los desarrolladores de aplicaciones contra las complejidades de los subsistemas. Los desarrolladores necesitan estar familiarizados solamente con las operaciones del objeto fachada, en vez de tener que conocer las operaciones más detalladas de todo el subsistema completo. La implementación detrás de la fachada puede modificarse sin necesidad de realizar cambios en los clientes.

En el paquete `java.net`, un objeto de la clase `URL` es un objeto fachada. Este objeto contiene una referencia a un objeto `InetAddress` que especifica la dirección IP del equipo host. El objeto fachada URL también hace referencia a un objeto de la clase `URLStreamHandler`, el cual abre la conexión del URL. El objeto cliente que utiliza el objeto fachada `URL` accede a los objetos `InetAddress` y `URLStreamHandler` a través del objeto fachada. Sin embargo, el objeto cliente no sabe cómo los objetos detrás del objeto fachada `URL` cumplen con sus responsabilidades.

18.12.3 Patrones de arquitectura

Los patrones de diseño permiten a los desarrolladores diseñar partes específicas de subsistemas, como el abstraer los instanciamientos de objetos o el agregar clases a estructuras más grandes. Los patrones de diseño también promueven el poco acoplamiento entre los objetos. Los *patrones de arquitectura* promueven el poco acoplamiento entre los subsistemas. Estos patrones especifican cómo deben interactuar los sistemas entre sí.[2] Vamos a presentar los populares patrones arquitectónicos Modelo-vista-controlador y Niveles o capas (Layers).

MVC

Considere el diseño de un editor de texto simple. En este programa, el usuario introduce texto desde el teclado y da formato a este texto mediante el ratón. Nuestro programa almacena este texto y la información del formato en una serie de estructuras de datos, y después muestra esta información en pantalla para que el usuario pueda leer lo que ha introducido.

Este programa se adhiere al *patrón de arquitectura Modelo-vista-controlador* (MVC), el cual separa los datos de la aplicación (contenidos en el *modelo*) de los componentes de presentación gráfica (la *vista*) y la lógica de procesamiento de la entrada (el *controlador*).[3] En la figura 18.25 se muestran las relaciones entre los componentes en el MVC.

El controlador implementa la lógica para procesar la entrada del usuario. El modelo contiene los datos de la aplicación y la vista presenta los datos almacenados en el modelo. Cuando un usuario proporciona datos de entrada, el controlador modifica el modelo en base a la entrada recibida. El modelo contiene los datos de la aplicación. Con respecto al ejemplo del editor de texto, el modelo podría contener solamente los caracteres que conforman el documento. Cuando el modelo cambia, notifica a la vista acerca del cambio, para que ésta pueda actualizar su presentación con los datos modificados. La vista en un procesador de palabras podría mostrar los caracteres utilizando un tipo de letra específico, con un tamaño específico, etcétera.

El MVC no restringe a una aplicación a una sola vista o a un solo controlador. En un programa más sofisticado (como un procesador de palabras), podría haber dos vistas de un modelo de documento. Una vista podría mostrar un bosquejo del documento y el otro podría mostrar el documento completo. El procesador de palabras podría también implementar varios controladores: uno para manejar la entrada del teclado y otro para manejar las selecciones del ratón. Si cualquiera de los controladores realiza un cambio en el modelo, tanto la vista del bosquejo como la ventana de vista previa de impresión mostrarán el cambio inmediatamente, cuando el modelo notifique los cambios a todas las vistas.

Otro beneficio clave para el patrón de arquitectura MVC es que los desarrolladores pueden modificar cada componente de manera individual, sin tener que modificar los demás componentes. Por ejemplo, los desarrolladores podrían modificar la vista que muestra el documento en línea, y no tendrían que modificar ni el modelo ni las demás vistas o controladores.

2. R. Hartman. "Building on Patterns," *Application Development Trends* Mayo 2001: 19 a 26.
3. En la sección 13.17 también hablamos sobre la arquitectura Modelo-vista-controlador y su relevancia con el ejemplo práctico de simulación del elevador.

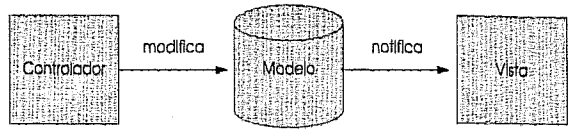


Figura 18.25 Arquitectura Modelo-vista-controlador.

Niveles o capas

Considere el diseño de la figura 18.26, el cual presenta la estructura básica de una *aplicación de tres niveles*, en la que cada nivel contiene un componente único del sistema.

El *nivel de información* (también conocido como el "nivel inferior") mantiene los datos para la aplicación, guardando generalmente los datos en una base de datos. El nivel de información para una tienda en línea puede contener información sobre los productos, como las descripciones, precios, cantidades en almacén y la información del cliente, como los nombres de usuarios, direcciones de facturación y números de tarjetas de crédito.

El *nivel medio* actúa como un intermediario entre el nivel de información y el nivel cliente. El nivel medio procesa las peticiones del nivel cliente, y lee los datos de (escribe los datos hacia) la base de datos. El nivel medio entonces procesa los datos del nivel de información y presenta el contenido al nivel cliente. Este procesamiento es la *lógica de negocios* de la aplicación, la cual se encarga de las tareas como recuperar los datos del nivel de información, asegurar que los datos sean confiables antes de actualizar la base de datos y presentar los datos al nivel cliente. Por ejemplo, la lógica de negocios asociada con el nivel medio para la tienda en línea puede verificar la tarjeta de crédito de un cliente con el emisor de la tarjeta de crédito antes de que el almacén envíe el pedido del cliente. Esta lógica de negocios podría entonces almacenar (o recuperar) la información de crédito en la base de datos y notificar al nivel cliente que la verificación tuvo éxito.

El *nivel cliente* (también conocido como el "nivel superior") es la interfaz de usuario de la aplicación, como un navegador Web estándar. Los usuarios interactúan directamente con la aplicación a través de la interfaz de usuario. El nivel cliente interactúa con el nivel medio para hacer peticiones y recuperar datos del nivel de información. El nivel cliente entonces muestra los datos recuperados del nivel medio.

En la figura 18.26 se muestra una implementación del *patrón de arquitectura Niveles*, el cual divide la funcionalidad en *niveles* separados. Cada nivel contiene un conjunto de responsabilidades del sistema y depende

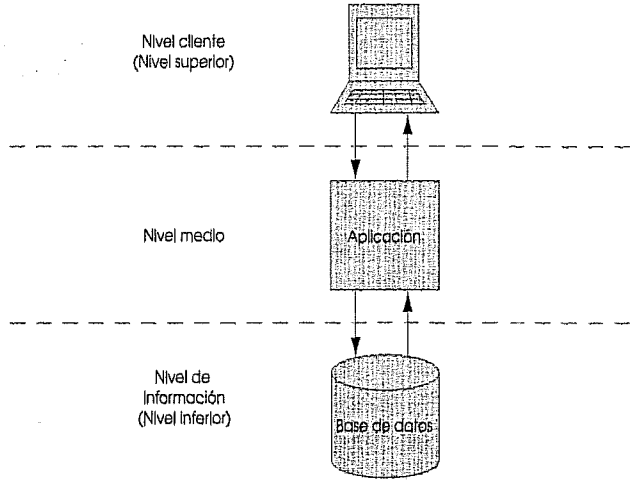


Figura 18.26 Modelo de aplicación de tres niveles.

http://librosysolucionarios.net

de los servicios del siguiente nivel inferior, solamente. En la figura 18.26, cada nivel corresponde a una capa. Este patrón de arquitectura es útil, ya que un diseñador puede modificar una capa sin tener que modificar las demás. Por ejemplo, un diseñador podría modificar el nivel de información en la figura 18.26 para incluir un producto de bases de datos específico, pero el diseñador no tendría que modificar el nivel cliente o el nivel medio.

18.12.4 Conclusión

En esta sección "Descubrimiento de los patrones de diseño" hablamos sobre cómo los paquetes `java.io` y `java.net` aprovechan los patrones de diseño específicos, y cómo los desarrolladores pueden integrar los patrones de diseño con aplicaciones de redes/archivos en Java. También presentamos los patrones de arquitectura Modelo-vista-controlador y Niveles o capas (Layers), que asignan la funcionalidad del sistema a subsistemas separados. Estos patrones facilitan a los desarrolladores el diseño de un sistema. En la sección "Descubrimiento de los patrones de diseño" 22.12, concluiremos con nuestra presentación de los patrones de diseño al describir los patrones utilizados en el paquete `java.util`.

RESUMEN

- Java proporciona sockets de flujo y sockets de datagrama. Con los sockets de flujo, un proceso establece una conexión con otro proceso. Mientras la conexión está en pie, los datos fluyen entre los procesos en flujos. Se dice que los sockets de flujo proporcionan un servicio orientado a la conexión. El protocolo utilizado para la transmisión es el popular TCP (Protocolo de control de transmisión).
- Con los sockets de datagrama se transmiten paquetes individuales de información. UDP (Protocolo de datagramas de usuario) es un servicio sin conexión que no garantiza que los paquetes lleguen en un orden específico. Los paquetes pueden perderse, duplicarse e incluso pueden llegar fuera de secuencia. Se requiere de programación adicional por parte del programador para hacer frente a estos problemas.
- El protocolo HTTP (Protocolo de transferencia de hipertexto) que forma la base de la Web, utiliza URIs (Identificadores uniformes de recursos) para identificar datos en Internet. Los URIs comunes representan archivos o directorios y pueden representar tareas complejas tales como búsquedas en bases de datos y en Internet. Un URI que representa a un documento se conoce como URL (Localizador uniforme de recursos).
- A menudo, los navegadores Web restringen a un applet de manera que sólo pueda comunicarse con el equipo desde el que se descargó originalmente.
- Un objeto `HashMap` almacena los pares clave/valor. Un programa utiliza una clave para almacenar y recuperar un valor asociado en el objeto `HashMap`. El método `put` de `HashMap` toma dos argumentos (una clave y su valor asociado) y coloca el valor en el objeto `HashMap`, en una ubicación determinada por la clave. El método `get` de `HashMap` toma un argumento (una clave) y recupera el valor (como una referencia `Object`) asociado con la clave.
- Un objeto `Vector` es un arreglo de objetos `Object` que puede cambiar su tamaño en forma dinámica. El método `add` de `Vector` agrega un nuevo elemento al final del objeto `Vector`.
- El método `getAppletContext` de `Applet` devuelve una referencia a un objeto `AppletContext` que representa el entorno de un applet (es decir el navegador en el que se ejecuta el applet). El método `showDocument` de `AppletContext` recibe un `URL` como argumento y lo pasa al objeto `AppletContext` (es decir, el navegador), el cual muestra el recurso Web asociado con ese `URL`. Una segunda versión de `showDocument` permite a un applet especificar el marco de destino en el que se mostrará un recurso Web. Los marcos de destino especiales son `_blank` (se muestra en una nueva ventana del navegador Web), `_self` (se muestra en el mismo marco que el applet) y `_top` (se eliminan los marcos actuales y después se muestra en la ventana actual).
- El método `setPage` de `JEditorPane` descarga el documento especificado por su argumento y lo muestra en el objeto `JEditorPane`.
- Por lo general, un documento HTML contiene hipervínculos (texto, imágenes o componentes de la GUI) que, cuando se hace clic sobre ellos, proporcionan un vínculo hacia otro documento en Web. Si se muestra un documento HTML en un objeto `JEditorPane` y el usuario hace clic en un hipervínculo, el objeto `JEditorPane` genera un evento `HyperlinkEvent` y notifica a todos los objetos `HyperlinkListener` registrados acerca de ese evento.
- El método `getEventType` de `HyperlinkEvent` determina el tipo del evento. `HyperlinkEvent` contiene una clase anidada `EventType`, la cual declara tres tipos de eventos de hipervínculo: `ACTIVATED` (se hizo clic en un hipervínculo), `ENTERED` (se puso el ratón sobre un hipervínculo) y `EXITED` (se alejó el ratón de un hipervínculo). El método `getURL` de `HyperlinkEvent` obtiene el `URL` representado por el hipervínculo.

- Las conexiones basadas en sockets se manejan con objetos `Socket`.
- Un objeto `ServerSocket` establece el puerto en el que un servidor espera las conexiones de los clientes. El segundo argumento para el constructor de `ServerSocket` es el número de conexiones que pueden esperar en una cola para conectarse al servidor. Si la cola de clientes está llena, se rechazan las demás conexiones de los clientes. El método `accept` de `ServerSocket` espera indefinidamente (es decir, bloquea) una conexión de un cliente y devuelve un objeto `Socket` cuando se establece una conexión.
- Los métodos `getOutputStream` y `getInputStream` de `Socket` obtienen referencias a los objetos `OutputStream` e `InputStream` asociadas con un objeto `Socket`, respectivamente. El método `close` de `Socket` termina una conexión.
- Un objeto `Socket` conecta un cliente con un servidor, especificando el nombre del servidor y el número de puerto al crear el objeto `Socket`. Un intento de conexión fallido lanza una excepción `IOException`.
- El método `getByName` de `InetAddress` devuelve un objeto `InetAddress` que contiene el nombre de host del equipo para el que se especifica el nombre de host o una dirección IP como argumento. EL método `getLocalHost` de `InetAddress` devuelve un objeto `InetAddress` que contiene el nombre de host del equipo local que ejecuta el programa.
- La transmisión orientada a la conexión es como el sistema telefónico; usted marca y recibe una conexión al teléfono de la persona con la que desea comunicarse. La conexión se mantiene todo el tiempo que dure su llamada telefónica, incluso aunque usted no esté hablando.
- La transmisión sin conexión mediante datagramas es un proceso similar a la manera en que el correo se transporta mediante el servicio postal. Un mensaje extenso que no cabe en un sobre puede dividirse en varias piezas separadas que se colocan en sobres separados, numerados en forma secuencial. Cada una de las cartas se envía entonces por correo al mismo tiempo. Las cartas podrían llegar en orden, desordenadas o tal vez podrían no llegar.
- Los objetos `DatagramPacket` almacenan paquetes de datos para enviar o almacenar paquetes de datos recibidos por una aplicación. Los objetos `DatagramSocket` envían y reciben objetos `DatagramPacket`.
- El constructor de `DatagramSocket` que no toma argumentos enlaza la aplicación a un puerto elegido por la computadora en la que se ejecuta el programa. El constructor de `DatagramSocket` que toma un número de puerto como argumento entero, enlaza la aplicación al puerto especificado. Si un constructor de `DatagramSocket` no puede enlazar la aplicación a un puerto, se produce una excepción `SocketException`. El método `receive` de `DatagramSocket` hace un bloqueo (espera) hasta que llega un paquete y después lo almacena en su argumento.
- El método `getAddress` de `DatagramPacket` devuelve un objeto `InetAddress` que contiene información acerca del equipo host desde el cual se envió el paquete. El método `getPort` devuelve un entero que especifica el número de puerto a través del cual el equipo host envió el objeto `DatagramPacket`. El método `getLength` devuelve un entero que representa el número de bytes de datos en un objeto `DatagramPacket`. El método `getData` devuelve un arreglo `byte` que contiene los datos en un objeto `DatagramPacket`.
- El constructor de `DatagramPacket` para enviar un mensaje toma cuatro argumentos; el arreglo de bytes a enviar, el número de bytes a enviar, la dirección del cliente al que se enviará el paquete y el número de puerto en donde el cliente estará esperando recibir los paquetes.
- El método `send` de `DatagramSocket` envía un objeto `DatagramPacket` a la red.
- Si ocurre un error al recibir o enviar un objeto `DatagramPacket`, se produce una excepción `IOException`.
- El proceso de leer datos de un objeto `Socket` es una llamada de bloqueo: el subproceso actual se coloca en estado bloqueado mientras el subproceso espera a que la operación de lectura se complete. El método `setSoTimeout` especifica que, si no se reciben datos en el número dado de milisegundos, el objeto `Socket` debe producir una excepción `InterruptedIOException`, la cual puede ser atrapada por el subproceso actual y luego seguir con la ejecución. Esto evita que el subproceso actual se bloquee indefinidamente, si no hay más datos disponibles que provengan del objeto `Socket`.
- La transmisión múltiple es una manera efectiva de enviar datos a muchos clientes sin incurrir en la sobrecarga producida al transmitir esos datos a todos los hosts en Internet. Mediante el uso de la transmisión múltiple, una aplicación puede "publicar" objetos `DatagramPacket` para entregarlos a las aplicaciones suscriptoras. Una aplicación transmite objetos `DatagramPackets` en forma múltiple, enviando los objetos `DatagramPackets` a una dirección de transmisión múltiple (multicast), la cual es una dirección IP en el rango de 224.0.0.0 a 239.255.255.255, reservada para la transmisión múltiple.

http://librosysolucionarios.net

- Los clientes que desean recibir estos objetos `DatagramPacket` pueden unirse al grupo de transmisión múltiple que recibirá los objetos `DatagramPacket` publicados en la dirección de transmisión múltiple.
- Los objetos `DatagramPacket` de transmisión múltiple no son confiables; no se garantiza que los paquetes lleguen a cualquier destino. Además, el orden en el que los clientes reciben los datagramas no se garantiza.
- El constructor de `MulticastSocket` toma como argumento el puerto al que debe conectarse el objeto `MulticastSocket` para recibir los objetos `DatagramPacket` entrantes.
- El método `joinGroup` de la clase `MulticastSocket` toma como argumento el objeto `InetAddress` del grupo de transmisión múltiple al que se va a unir. El método `receive` de la clase `MulticastSocket` lee un objeto `DatagramPacket` entrante desde una dirección de transmisión múltiple.
- En versiones anteriores a J2SE 1.4, para permitir la comunicación entre dos o más clientes con el servidor en forma concurrente, utilizando sockets, generalmente se utiliza un servidor con subprocesamiento múltiple. Un subproceso servidor maneja las operaciones de E/S de cada cliente, lo cual produce una excesiva sobrecarga en los subprocesos y, por ende, reduce la escalabilidad. La J2SE 1.4 resuelve este problema mediante la introducción del mecanismo de E/S sin bloqueo.
- Mediante la tecnología de E/S sin bloqueo, un servidor puede interactuar con varias conexiones de clientes en forma simultánea, sin necesidad de crear un subproceso para cada conexión. La comunicación entre el servidor y el cliente se lleva a cabo a través de *canales*, los cuales pueden trabajar en modo sin bloqueo.
- La tecnología sin bloqueo se implementa mediante objetos `SelectableChannel` en conjunto con objetos `Selector`. Ambas clases son parte del paquete `java.nio.channels`.
- Un objeto `SelectableChannel` representa un canal de comunicación que puede administrar varios canales de E/S en forma simultánea, a lo que se le conoce también como multiplexión, cuando trabaja en conjunto con un objeto `Selector`.
- Un objeto `Selector`, que es un multiplexor de objetos `SelectableChannel`, permite comprobar los canales listos en forma simultánea.
- Un objeto `ServerSocketChannel` realiza la misma tarea que `ServerSocket`: escucha las conexiones de sockets. La principal diferencia entre `ServerSocket` y `ServerSocketChannel` es que `ServerSocketChannel` puede trabajar en modo sin bloqueo.
- Un objeto `SelectionKey` representa el registro de un canal con un selector.
- El método `selectedKeys` de la clase `Selector` devuelve un objeto `Set` que representa la clave seleccionada por el selector. Todas las claves seleccionadas son de aceptación, lectura o escritura.
- El método `write` de la clase `SocketChannel` toma un objeto `ByteBuffer`, el cual especifica el búfer desde el cual se van a escribir los valores `byte`, y devuelve un `int` que indica cuántos valores `byte` se van a escribir en el canal.
- El método `read` de la clase `SocketChannel` toma un objeto `ByteBuffer`, el cual especifica el búfer al cual se van a leer los valores `byte`, y devuelve un `int` que indica cuántos valores `byte` se van a leer del canal.

TERMINOLOGÍA

abrir un socket
`accept`, método de `ServerSocket`
`accept`, método de `ServerSocketChannel`
aceptar una conexión
`add`, método de `Vector`
`allocate`, método `static` de `ByteBuffer`
API de seguridad de Java
aplicaciones de colaboración
`AppletContext`, interfaz
ByteBuffer
canales
cerrar la conexión
cliente
cliente se conecta a un servidor
`close`, método de `Selector`
`close`, método de `ServerSocketChannel`
`close`, método de `Socket`
`ClosedByInterruptException`
comunicaciones basadas en sockets
conectar a un puerto
conectar a un sitio Web
conexión
`configureBlocking`, método
`connect`, método de `SocketChannel`
`ConnectException`, clase
`channel`, método de `SelectionKey`
CharBuffer
datagrama
`DatagramPacket`, clase
`DatagramSocket`, clase
dirección de transmisión múltiple
dirección IP
E/S sin bloqueo
enlazar a un puerto
esperar un paquete
esperar una conexión
`flip`, método de `ByteBuffer`
`get`, método de la clase `HashMap`
`getAddress`, método de `DatagramPacket`
`getAppletContext`, método de `Applet`
`getByName`, método de `InetAddress`
`getData`, método de `DatagramPacket`
`getEventType`, método de `HyperlinkEvent`
`getInputStream`, método de `Socket`
`getLength`, método de `DatagramPacket`
`getLocalHost`, método `InetAddress`
`getOutputStream`, método de `Socket`
`getPort`, método de `DatagramPacket`
`getURL`, método de `HyperlinkEvent`
grupo de transmisión múltiple
`HashMap`, clase
host
`HyperlinkEvent`, clase
`HyperlinkEvent.EventTyp`, clase anidada
`HyperlinkListener`, interfaz
`hyperlinkUpdate`, método
`InetAddress`, clase
`InterruptedIOException`, clase
`IOException`, clase
`isAcceptable`, método de `SelectionKey`
`isOpen`, método de `SocketChannel`
`isReadable`, método de `SelectionKey`
`java.net`, paquete
`java.nio.channels`, paquete
`JEditorPane`, clase
`joinGroup`, método de `MulticastSocket`
`leaveGroup`, método de `MulticastSocket`
leer de un socket
longitud de paquete
`MalformedURLException`, clase
`MulticastSocket`, clase
Navegador Web
número de puerto en un servidor
`open`, método de `ServerSocketChannel`
`open`, método de `SocketChannel`
`openSelector`, método de `SelectorProvider`
paquete
paquetes perdidos
paquetes duplicados
paquetes fuera de secuencia
pares clave/valor
petición de conexión
programación en red
`provider`, método de `SelectorProvider`
puerto
punto de negociación
`put`, método de `HashMap`
`read`, método de `SocketChannel`
`receive`, método de `DatagramSocket`
rechazar una conexión
redes de computadoras
registrar un número de puerto disponible
relación cliente/servidor
`select`, método de `Selector`
`SelectableChannel`
`selectedKeys`, método de `Selector`
`SelectionKey`
`SelectionKey.OP_ACCEPT`, constante
`SelectionKey.OP_READ`, constante
`Selector`
`selector`, método de `SelectionKey`
`send`, método de `DatagramSocket`
`ServerSocket`, clase
`ServerSocketChannel`
servicio orientado a la conexión
servicio sin conexión
servidor
servidor con subprocesamiento múltiple
servidor Web
`setPage`, método de `JEditorPane`
`setSoTimeout`, método de `Socket`
`showDocument`, método de `AppletContext`
sistemas computacionales heterogéneos
socket
sockets de datagrama
socket de flujo
socket del lado cliente
socket del lado servidor
`Socket`, clase
`socket`, método de `ServerSocketChannel`
`socket`, método de `SocketChannel`
`SocketException`
TCP (Protocolo de control de transmisión)
`toString`, método de `CharBuffer`
transmisión múltiple (multicast)
transmisión sin conexión mediante datagramas
UDP (Protocolo de datagramas de usuario)
`UnknownHostException`
URI (Identificador uniforme de recursos)
URL (Localizador uniforme de recursos)
`URL`, clase
`Vector`, clase
World Wide Web
`write`, método de `SocketChannel`

EJERCICIOS DE AUTOEVALUACIÓN

18.1 Complete las siguientes oraciones:

a) La excepción _____ ocurre cuando se produce un error de entrada/salida al cerrar un socket.
b) La excepción _____ ocurre cuando un nombre de host indicado por un cliente no se puede resolver a una dirección.
c) Si un constructor de `DatagramSocket` no puede establecer un objeto `DatagramSocket` apropiadamente, se produce una excepción del tipo _____.
d) Muchas de las clases para trabajo en red de Java están contenidas en el paquete _____.
e) El método _____ enlaza la aplicación a un puerto para la transmisión de datagramas.
f) Un objeto de la clase _____ contiene una dirección IP.

http://librosysolucionarios.net

g) Los dos tipos de sockets que vimos en este capítulo son _____ y _____.
h) El acrónimo URL significa _____.
i) El acrónimo URI significa _____.
j) El protocolo clave que forma la base de la World Wide Web es _____.
k) El método _____ de `AppletContext` recibe un objeto URL como argumento y muestra en un navegador el recurso Web asociado con ese URL.
l) El método `getLocalHost` devuelve un objeto _____ que contiene el nombre de host local de la computadora en la que se está ejecutando el programa.
m) El método _____ de `MulticastSocket` suscribe un `MulticastSocket` a un grupo de transmisión múltiple.
n) El constructor de `URL` determina si su argumento de cadena es un URL válido. De ser así, el objeto `URL` se inicializa con esa ubicación; en caso contrario, se produce una excepción _____.

18.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Mediante la transmisión múltiple se transmiten objetos `DatagramPacket` a todos los hosts en Internet.
b) UDP es un protocolo orientado a la conexión.
c) Con los sockets de flujo, un proceso establece una conexión con otro proceso.
d) Un servidor espera en un puerto las conexiones de un cliente.
e) La transmisión de paquetes con datagramas a través de una red es un proceso confiable; se garantiza que los paquetes lleguen en secuencia.
f) Por razones de seguridad, muchos navegadores Web como Netscape permiten a los applet de Java realizar el procesamiento de archivos solamente en los equipos en los que se ejecutan.
g) A menudo, los navegadores Web restringen a un applet de manera que sólo pueda comunicarse con el equipo desde el que se descargó originalmente.
h) Las direcciones IP desde `224.0.0.0` hasta `239.255.255.255` están reservadas para la transmisión múltiple.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

18.1 a) `IOException`. b) `UnknownHostException`. c) `SocketException`. d) `java.net`. e) `DatagramSocket`. f) `InetAddress`. g) Sockets de flujo, sockets de datagrama. h) Localizador uniforme de recursos. i) Identificador uniforme de recursos. j) HTTP. k) `showDocument`. l) `InetAddress`. m) `joinGroup`. n) `MalformedURLException`.

18.2 a) Falso; la transmisión múltiple envía objetos `DatagramPacket` solamente a los hosts que se hayan unido al grupo de transmisión múltiple. b) Falso; UDP es un protocolo sin conexión y TCP es un protocolo orientado a la conexión. c) Verdadero. d) Verdadero. e) Falso; los paquetes podrían perderse y podrían llegar desordenados. f) Falso; la mayoría de los navegadores evita que los applet realicen un procesamiento de archivos en el equipo cliente. g) Verdadero. h) Verdadero.

EJERCICIOS

18.3 Indique la diferencia entre los servicios de red orientados a la conexión y los servicios sin conexión.

18.4 ¿Cómo determina un cliente el nombre de host del equipo cliente?

18.5 ¿Bajo qué circunstancias se lanzaría una excepción `SocketException`?

18.6 ¿Cómo puede un cliente obtener una línea de texto de un servidor?

18.7 Describa cómo se conecta un cliente con un servidor.

18.8 Describa cómo un servidor envía datos a un cliente.

18.9 Describa cómo operar un servidor para recibir una petición de conexión basada en flujo de un solo cliente.

18.10 Describa cómo preparar a un servidor para que reciba peticiones de conexión de varios clientes, si cada cliente que se conecte debe procesarse en forma concurrente con los demás clientes conectados.

18.11 ¿Cómo escucha un servidor las conexiones de sockets basadas en flujo en un puerto?

18.12 ¿Qué es lo que determina cuántas peticiones de conexión de los clientes pueden esperar en una cola para conectarse con un servidor?

18.13 Según lo descrito en el texto, ¿cuáles son las razones que podrían ocasionar que un servidor rechace una petición de conexión de un cliente?

http://librosysolucionarios.net

18.14 Use una conexión de socket para permitir a un cliente especificar un nombre de archivo, y haga que el archivo envíe el contenido del mismo o indique que el archivo no existe.

18.15 Modifique el ejercicio 18.14 para permitir al cliente modificar el contenido del archivo y enviarlo de vuelta al servidor para que lo almacene. El usuario puede editar el archivo en un objeto `JTextArea` y después hacer clic en un botón *guardar cambios* para enviar el archivo de vuelta al servidor.

18.16 Modifique el programa de la figura 18.2 para permitir que los usuarios agreguen sus propios sitios a la lista, y que también los eliminen de la lista.

18.17 Los servidores con subprocesamiento múltiple son bastante populares hoy en día, especialmente debido al uso cada vez mayor de servidores con multiprocesamiento. Modifique la aplicación de servidor simple presentada en la sección 18.6, para que sea un servidor con subprocesamiento múltiple. Después utilice varias aplicaciones cliente y haga que cada una de ellas se conecte al servidor en forma simultánea. Use un objeto `Vector` para almacenar los subprocesos cliente. `Vector` proporciona varios métodos que pueden utilizarse en este ejercicio. El método `size` determina el número de elementos en un objeto `Vector`. El método `get` devuelve el elemento (como una referencia `Object`) en la ubicación especificada por su argumento. El método `add` coloca su argumento al final del objeto `Vector`. El método `remove` elimina su argumento del objeto `Vector`. El método `lastElement` devuelve una referencia `Object` al último elemento que se haya insertado en el objeto `Vector`.

18.18 En el texto presentamos un programa de tres en raya, controlado por un servidor con subprocesamiento múltiple. Desarrolle un programa de damas que se modele con base en el programa de tres en raya. Los dos usuarios deberán alternar sus movimientos. Su programa debe mediar los movimientos de los jugadores, determinando el turno de cada quién y permitiendo sólo movimientos válidos. Los mismos jugadores determinarán cuando el juego se haya acabado.

18.19 Desarrolle un programa para jugar ajedrez, modelado en base al programa de damas del ejercicio 18.18.

18.20 Desarrolle un programa para jugar cartas estilo Blackjack, en el que la aplicación servidor reparta las cartas a cada uno de los applet cliente. El servidor deberá repartir cartas adicionales (según las reglas del juego) a cada jugador, según sea requerido.

18.21 Desarrolle un programa para jugar cartas estilo Póquer, en el que la aplicación servidor reparta las cartas a cada uno de los applet cliente. El servidor deberá repartir cartas adicionales (según las reglas del juego) a cada jugador, según sea requerido.

18.22 *(Modificaciones al programa de tres en raya con subprocesamiento múltiple.)* Los programas de las figuras 18.8 y 18.9 implementan una versión cliente/servidor con subprocesamiento múltiple del juego Tres en raya. Nuestro objetivo al desarrollar este juego fue demostrar el uso de un servidor con subprocesamiento múltiple que pudiera procesar varias conexiones de clientes al mismo tiempo. El servidor en el ejemplo realmente es un mediador entre los dos applet cliente; se asegura que cada movimiento sea válido y que cada cliente se mueva en el orden apropiado. El servidor no determina quién ganó o perdió, o si hubo un empate. Además, no existe la capacidad de permitir que se inicie un juego nuevo, o de terminar un juego existente.
A continuación se muestra una lista de modificaciones sugeridas a las figuras 18.8 y 18.9:
a) Modificar la clase `ServidorTresEnRaya` para probar si se ganó, perdió o empató en cada movimiento del juego. Enviar un mensaje a cada applet cliente que indique el resultado del juego cuando éste termine.
b) Modificar la clase `ClienteTresEnRaya` para mostrar un botón que, al hacer clic sobre él, el cliente pueda jugar otro juego. El botón deberá habilitarse sólo cuando se complete un juego. Observe que las clases `ClienteTresEnRaya` y `ServidorTresEnRaya` deben modificarse para restablecer el tablero y toda la demás información de estado. Además, el otro `ClienteTresEnRaya` debe ser notificado cuando un nuevo juego esté por empezar, para que su tablero y su estado puedan restablecerse.
c) Modificar la clase `ClienteTresEnRaya` para mostrar un botón que, al hacer clic sobre él, el cliente pueda terminar el programa en cualquier momento. Cuando el usuario haga clic en el botón, el servidor y el otro cliente deberán ser notificados. Después el servidor deberá esperar una conexión de otro cliente, para que pueda empezar un juego nuevo.
d) Modifique las clases `ClienteTresEnRaya` y `ServidorTresEnRaya` de manera que el ganador de un juego pueda seleccionar la pieza X u O para el siguiente juego. Recuerde: la X siempre va primero.
e) Si le gusta ser ambicioso, permita a un cliente jugar contra el servidor mientras éste espera la conexión de otro cliente.

18.23 *(Tres en raya con subprocesamiento múltiple, en 3-D.)* Modifique el programa de Tres en raya cliente/servidor con subprocesamiento múltiple, para implementar una versión tridimensional del juego, de 4 × 4 × 4. Implemente la aplicación servidor para que sea el intermediario entre los dos clientes. Muestre el tablero tridimensional como cua-

tro tableros que contienen cuatro filas y cuatro columnas cada uno. Si le gusta ser ambicioso, trate de hacer las siguientes modificaciones:

a) Dibuje el tablero en forma tridimensional.
b) Permita al servidor probar si hay un ganador, un perdedor o si fue empate. ¡Cuidado! ¡Hay muchas posibles maneras de ganar en un tablero de 4 × 4 × 4!

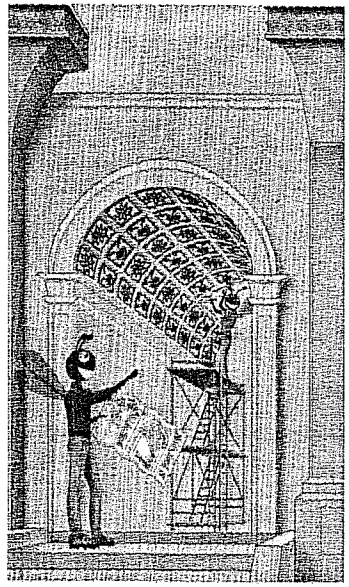
18.24 *(Código Morse en red.)* Modifique su solución al ejercicio 11.27 para permitir que dos applet se envíen entre sí mensajes en código Morse, a través de una aplicación servidor con subprocesamiento múltiple. Cada applet deberá permitir al usuario escribir caracteres normales en objetos `JTextArea`, traducir los caracteres en Código Morse y enviar el mensaje codificado a través del servidor, al otro cliente. Cuando se reciban los mensajes, deberán ser decodificados y mostrados como caracteres normales y como Código Morse. El applet debe tener dos objetos `JTextArea`: uno para mostrar los mensajes de los otros clientes y uno para escribir.

19

Multimedia: Imágenes, animación y audio

Objetivos

- Aprender a obtener y mostrar imágenes.
- Crear animaciones a partir de secuencias de imágenes.
- Crear mapas de imágenes.
- Obtener sonidos, reproducirlos, hacer que se reproduzcan indefinidamente y detenerlos mediante el uso de un objeto `AudioClip`.



La rueda que rechina más fuerte... es la que requiere la grasa.
John Billings (Henry Wheeler Shaw)

Utilizaremos una señal que he probado que tiene gran alcance y es fácil de gritar. ¡Yajuuuu!
Zane Grey

Hay un movimiento de vaivén natural en un pez dorado.
Walt Disney

Entre el movimiento y el acto cae la sombra.
Thomas Stearns Eliot

http://librosysolucionarios.net

Plan general

19.1 Introducción
19.2 Cómo cargar, mostrar y escalar imágenes
19.3 Animación de una serie de imágenes
19.4 Mapas de imágenes
19.5 Carga y reproducción de clips de audio
19.6 Recursos en Internet y World Wide Web
19.7 (Ejemplo práctico opcional) Acerca de los objetos: Animación y sonido en la vista

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Sección especial: Proyectos de multimedia retadores

19.1 Introducción

Bienvenido a lo que podría ser la mayor revolución en la historia de la industria computacional. Todos los que entramos al campo hace décadas, estábamos interesados en usar las computadoras principalmente para realizar cálculos aritméticos a grandes velocidades. A medida que el campo de la computación va evolucionando, estamos empezando a darnos cuenta de que las herramientas de manipulación de datos de las computadoras son ahora igualmente importantes. El "atractivo" de Java es *multimedia* (el uso de *sonido, imágenes, gráficos* y *vídeo* para hacer que las aplicaciones "cobren vida"). Hoy en día, muchas personas consideran que el vídeo de color bidimensional es lo "último" en multimedia. Esperamos todo tipo de nuevas y excitantes aplicaciones tridimensionales. Los programadores de Java pueden hacer uso ya de la *API Java3D* para crear aplicaciones importantes con gráficos en 3D.

La programación de multimedia ofrece siempre nuevos retos. El campo es de por sí enorme, y crecerá rápidamente. La gente se está apresurando a equipar sus computadoras con capacidades multimedia. La mayoría de las nuevas computadoras que se venden actualmente están "listas para multimedia", con unidades de CD o DVD, tarjetas de audio e incluso con herramientas especiales de vídeo.

En la actualidad, a los usuarios ya no les basta con los gráficos bidimensionales; ahora desean gráficos de color tridimensionales, de alta resolución. El despliegue de imágenes tridimensionales verdaderas tal vez esté disponible dentro de la siguiente década. Imagine tener una televisión tridimensional, de alta resolución tipo cine. ¡Los eventos deportivos y de entretenimiento parecerán estar llevándose a cabo en la sala de su casa! Los estudiantes de medicina en todo el mundo verán cómo se realizan operaciones a miles de millas de distancia, como si estuvieran ocurriendo en la misma habitación. Las personas aprenderán a conducir con simuladores de conducción extremadamente realistas en sus hogares antes de ponerse al volante. Las posibilidades son emocionantes e interminables.

La multimedia demanda un poder computacional extraordinario. Hasta hace poco, no había computadoras a precios accesibles con ese tipo de poder. Los procesadores ultrarrápidos de la actualidad, como el SPARC Ultra de Sun Microsystems, Pentium e Itanium de Intel, Alpha de Hewlett-Packard y los procesadores de MIPS/Silicon Graphics (entre otros) hacen posible la multimedia con efectividad. Las industrias de la computación y la comunicación serán los principales beneficiarios de la revolución multimedia. Los usuarios estarán dispuestos a pagar por procesadores más veloces, memorias más grandes y anchos de banda de comunicación más amplios, que soporten las exigencias de las aplicaciones multimedia. Irónicamente, los usuarios tal vez no tengan que pagar más, ya que la feroz competencia en estas industrias hace que los precios bajen.

Necesitamos lenguajes de programación que faciliten la creación de aplicaciones multimedia. La mayoría de los lenguajes de programación no tienen herramientas multimedia integradas . Sin embargo, Java proporciona extensas características multimedia que le permitirán empezar a desarrollar poderosas aplicaciones multimedia inmediatamente.

En este capítulo presentamos varios ejemplos de las interesantes características multimedia que usted necesitará para crear útiles aplicaciones, incluyendo las siguientes:

1. Los fundamentos de la manipulación de imágenes.

http://librosysoluciones.net

2. Creación de animaciones refinadas.
3. Reproducción de archivos de audio mediante la interfaz `AudioClip`.
4. Creación de mapas de imágenes que pueden detectar cuando el cursor se encuentra sobre ellos, incluso sin hacer clic con el ratón.

Los ejercicios para este capítulo sugieren docenas de proyectos retadores e interesantes. Cuando creamos estos ejercicios, las ideas no dejaban de surgir. La multimedia impulsa la creatividad en formas que no hemos experimentado con las herramientas "convencionales" de computación.[1]

19.2 Cómo cargar, mostrar y escalar imágenes

Las herramientas multimedia de Java incluyen gráficos, imágenes, animaciones, sonidos y vídeo. Empezaremos nuestra discusión con las imágenes.

El applet (subprograma) de la figura 19.1 demuestra cómo cargar un objeto `Image` (paquete `java.awt`) y cómo cargar un objeto `ImageIcon` (paquete `javax.swing`). El applet muestra al objeto `Image` en su tamaño original y escalado a un tamaño mayor, utilizando dos versiones del método `drawImage` de `Graphics`. El applet también dibuja el objeto `ImageIcon`, utilizando el método `paintIcon` del icono. La clase `ImageIcon` es más fácil de usar que `Image`, ya que su constructor puede recibir argumentos de varios formatos distintos, incluyendo un arreglo `byte` que contiene los bytes de una imagen, un objeto `Image` ya cargado en memoria, una cadena que representa la ubicación de una imagen y un objeto `URL` que representa la ubicación de una imagen.

En las líneas 9 y 10 se declaran una referencia `Image` y una referencia `ImageIcon`, respectivamente. La clase `Image` es una clase `abstract`; por lo tanto, el applet no puede crear un objeto de la clase `Image` di-

```
1  // Fig. 19.1: CargarImagenYEscalar.java
2  // Cargar una imagen y mostrarla en su tamaño original y al doble de su
3  // tamaño original. Cargar y mostrar la misma imagen como un objeto ImageIcon.
4  import java.applet.Applet;
5  import java.awt.*;
6  import javax.swing.*;
7
8  public class CargarImagenYEscalar extends JApplet {
9     private Image logo1;
10    private ImageIcon logo2;
11
12    // cargar imagen cuando se cargue el applet
13    public void init()
14    {
15       logo1 = getImage( getDocumentBase(), "logo.gif" );
16       logo2 = new ImageIcon( "logo.gif" );
17    }
18
19    // mostrar imagen
20    public void paint( Graphics g )
21    {
```

Figura 19.1 Cargar y mostrar una imagen en un applet. (Parte 1 de 2.)

1. Las herramientas de multimedia de Java van más allá de las que se presentan en este capítulo. Entre estas herramientas se incluyen la API del marco de trabajos de medios de Java (Java Media Framework API), la API de sonido de Java (Java Sound API), la API 3D de Java (Java 3D API), la API de procesamiento avanzado de imágenes de Java (Java Advanced Imaging API), la API de síntesis de voz de Java (Java Speech API), la API 2D de Java (Java 2D API) y la API de E/S de imágenes de Java (Java Image I/O API). En la sección 19.6 se incluyen los URLs para la página inicial de cada una de estas APIs.

```
22          g.drawImage( logo1, 0, 0, this ); // dibujar la imagen original
23
24          // dibujar imagen para ajustarse a la anchura y la altura menos 120 píxeles
25          g.drawImage( logo1, 0, 120, getWidth(), getHeight() - 120, this );
26
27          // dibujar icono usando su método paintIcon
28          logo2.paintIcon( this, g, 180, 0 );
29       }
30
31    } // fin de la clase CargarImagenYEscalar
```



Figura 19.1 Cargar y mostrar una imagen en un applet. (Parte 2 de 2.)

rectamente; debe llamar a un método que haga que el contenedor de applet cargue y devuelva el objeto `Image` para usarlo en el programa. La clase `Applet` (la superclase directa de `JApplet`) proporciona el método `getImage` (línea 15 en el método `init`) para cargar un objeto `Image` en un applet. Esta versión de `getImage` toma dos argumentos: la ubicación del archivo de la imagen y el nombre de ese archivo. En el primer argumento, el método `getDocumentBase` de `Applet` devuelve un objeto `URL` que representa la ubicación de la imagen en Internet (o en su computadora, si el applet se cargó desde ahí). El programa supone que la imagen está almacenada en el mismo directorio que el archivo HTML que invocó al applet. El método `getDocumentBase` devuelve la ubicación del archivo HTML en Internet, como un objeto de la clase `URL`. El segundo argumento especifica el nombre de un archivo de imagen. Java soporta varios formatos de imágenes, incluyendo *GIF (Formato de intercambio de gráficos)*, *JPEG (Grupo unido de expertos en fotografía)* y *PNG (Gráficos portables de red)*. Los nombres de archivo para cada uno de estos tipos terminan con `.gif`, `.jpg` (o `.jpeg`) y `.png`, respectivamente.



Tip de portabilidad 19.1

La clase `Image` es una clase `abstract`; como resultado, los programas no pueden instanciar la clase `Image` para crear objetos. Para lograr una independencia de plataformas, la implementación de Java en cada plataforma proporciona su propia subclase de `Image` para almacenar información sobre las imágenes.

En la línea 15 se empieza a cargar la imagen del equipo local (o se empieza a descargar la imagen de Internet). Cuando la imagen es requerida por el programa, se carga en un subproceso de ejecución separado. Esto permite al programa continuar su ejecución mientras la imagen se carga. (*Nota:* Si el archivo de imagen no está disponible, el método `getImage` no indica un error.)

La clase `ImageIcon` no es una clase `abstract`; por lo tanto, un programa puede crear un objeto `ImageIcon`. La línea 16 en el método `init` crea un objeto `ImageIcon` que carga la misma imagen `logo.gif`. La clase `ImageIcon` proporciona varios constructores que permiten a los programas inicializar objetos `ImageIcon` con imágenes del equipo local, o con imágenes almacenadas en Internet.

El método `paint` del applet (líneas 20 a 29) muestra las imágenes. En la línea 22 se utiliza el método `drawImage` de `Graphics` para mostrar un objeto `Image`. El método `drawImage` recibe cuatro argumentos. El primer argumento es una referencia al objeto `Image` que se va a mostrar (`logo1`). Los argumentos segundo y tercero son las coordenadas *x* y *y* en las que se va a mostrar la imagen en el applet; las coordenadas indican la ubicación de la esquina superior izquierda de la imagen. El último argumento es una referencia a un objeto `ImageObserver`. Este argumento es importante cuando se muestran imágenes extensas que requieren de mucho tiempo para descargarse desde Internet. Es posible que un programa ejecute el código que muestre la imagen antes de que ésta se descargue completamente. El objeto `ImageObserver` es notificado para actualizar la imagen que se va a mostrar, a medida que la imagen termina de cargarse. Por lo general, el objeto `ImageObserver` es el objeto en el que el programa muestra la imagen. Un objeto `ImageObserver` puede ser cualquier objeto que implemente a la interfaz `ImageObserver`. La clase `Component` (una de las clases indirectas de `JApplet`) implementa a la interfaz `ImageObserver`. Por lo tanto, todos los objetos `Component` (incluyendo a nuestro applet) son objetos `ImageObserver`. Al ejecutar este applet, observe cuidadosamente cómo se van mostrando piezas de la imagen mientras ésta se carga. (*Nota:* En equipos más rápidos, tal vez no pueda percatarse de este efecto.)

En la línea 25 se utiliza otra versión del método `drawImage` de `Graphics` para mostrar una versión *escalada* de la imagen. Los argumentos cuarto y quinto especifican la *anchura* y *altura* de la imagen para mostrarla en pantalla. El método `drawImage` escala la imagen para que se ajuste a la anchura y altura especificadas. En este ejemplo, el cuarto argumento indica que la anchura de la imagen escalada debe ser igual a la anchura del applet, y el quinto argumento indica que la altura debe ser 120 píxeles menor que la altura del applet. La anchura y la altura del applet se determinan mediante llamadas a los métodos `getWidth` y `getHeight` (heredados de la clase `Component`).

En la línea 28 se utiliza el método `paintIcon` de `ImageIcon` para mostrar la imagen. Este método requiere cuatro argumentos: una referencia al objeto `Component` en el que se va a mostrar la imagen, una referencia al objeto `Graphics` que desplegará la imagen, la coordenada *x* de la esquina superior izquierda de la imagen y la coordenada *y* de la esquina superior izquierda de la imagen.

Si compara las dos técnicas para cargar y mostrar imágenes en este ejemplo, podrá ver que es más sencillo usar `ImageIcon`. Puede crear objetos de la clase `ImageIcon` directamente, y no hay necesidad de utilizar una referencia `ImageObserver` cuando se muestre la imagen. Por esta razón utilizaremos la clase `ImageIcon` durante el resto de ese capítulo. (*Nota:* El método `paintIcon` de la clase `ImageIcon` no permite escalar una imagen. Sin embargo, la clase `ImageIcon` proporciona el método `getImage`, el cual devuelve una referencia `Image` que el método `drawImage` de `Graphics` puede utilizar para mostrar una imagen escalada.)

19.3 Animación de una serie de imágenes

El siguiente ejemplo demuestra cómo animar una serie de imágenes almacenadas en un arreglo de objetos `ImageIcon`. La animación presentada en la figura 19.2 se designa como una subclase de `JPanel` (llamada `AnimadorLogo`) que puede adjuntarse a una ventana de aplicación o a un objeto `JApplet`. La clase `AnimadorLogo` también declara un método `main` (líneas 85 a 99) para ejecutar la animación como una aplicación. El método `main` declara una instancia de la clase `JFrame` y adjunta un objeto `LogoAnimator` para que el objeto `JFrame` muestre la animación.

```
1  // Fig. 19.2: AnimadorLogo.java
2  // Animación de una serie de imágenes.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
```

Figura 19.2 Animación de una serie de imágenes. (Parte 1 de 3.)

http://librosysolucionarios.net

```
6
7  public class AnimadorLogo extends JPanel implements ActionListener {
8
9     private final static String NOMBRE_IMAGEN = "deitel"; // nombre base de la imagen
10    protected ImageIcon imagenes[]; // arreglo de imágenes
11
12    private int imagenesTotales = 30;    // número de imágenes
13    private int imagenActual = 0;        // índice de imagen actual
14    private int retrasoAnimacion = 50;   // retraso en milisegundos
15    private int anchura;                 // anchura de la imagen
16    private int altura;                  // altura de la imagen
17
18    private Timer temporizadorAnimacion; // el objeto Timer controla la animación
19
20    // inicializar objeto AnimadorLogo cargando las imágenes
21    public AnimadorLogo()
22    {
23       imagenes = new ImageIcon[ imagenesTotales ];
24
25       // cargar imágenes
26       for ( int cuenta = 0; cuenta < imagenes.length; ++cuenta )
27          imagenes[ cuenta ] = new ImageIcon( getClass().getResource(
28             "imagenes/" + NOMBRE_IMAGEN + cuenta + ".gif" ) );
29
30       // en este ejemplo suponemos que todas las imágenes tienen la misma anchura y altura
31       anchura = imagenes[ 0 ].getIconWidth();    // obtener anchura de icono
32       altura = imagenes[ 0 ].getIconHeight();    // obtener altura de icono
33    }
34
35    // mostrar imagen actual
36    public void paintComponent( Graphics g )
37    {
38       super.paintComponent( g );
39
40       imagenes[ imagenActual ].paintIcon( this, g, 0, 0 );
41
42       // pasar a la siguiente imagen solamente si el temporizador se está ejecutando
43       if ( temporizadorAnimacion.isRunning() )
44          imagenActual = ( imagenActual + 1 ) % imagenesTotales;
45    }
46
47    // responder al evento del objeto Timer
48    public void actionPerformed( ActionEvent eventoAccion )
49    {
50       repaint(); // pintar el animador nuevamente
51    }
52
53    // iniciar o reiniciar la animación
54    public void iniciarAnimacion()
55    {
56       if ( temporizadorAnimacion == null ) {
57          imagenActual = 0;
58          temporizadorAnimacion = new Timer( retrasoAnimacion, this );
59          temporizadorAnimacion.start();
60       }
```

Figura 19.2 Animación de una serie de imágenes. (Parte 2 de 3.)


```
61       else // continuar desde la última imagen mostrada
62          if ( ! temporizadorAnimacion.isRunning() )
63             temporizadorAnimacion.restart();
64    }
65
66    // detener el temporizador de la animación
67    public void detenerAnimacion()
68    {
69       temporizadorAnimacion.stop();
70    }
71
72    // devolver el tamaño mínimo de la animación
73    public Dimension getMinimumSize()
74    {
75       return getPreferredSize();
76    }
77
78    // devolver el tamaño preferido de la animación
79    public Dimension getPreferredSize()
80    {
81       return new Dimension( anchura, altura );
82    }
83
84    // ejecutar la animación en un objeto JFrame
85    public static void main( String args[] )
86    {
87       AnimadorLogo animacion = new AnimadorLogo(); // crear el objeto AnimadorLogo
88
89       JFrame ventana = new JFrame( "Prueba del animador" ); // configurar ventana
90       ventana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
91
92       Container contenedor = ventana.getContentPane();
93       contenedor.add( animacion );
94
95       ventana.pack(); // hacer la ventana lo suficientemente grande para su GUI
96       ventana.setVisible( true );    // mostrar la ventana
97       animacion.iniciarAnimacion();  // comenzar la animación
98
99    } // fin del método main
100
101 } // fin de la clase AnimadorLogo
```

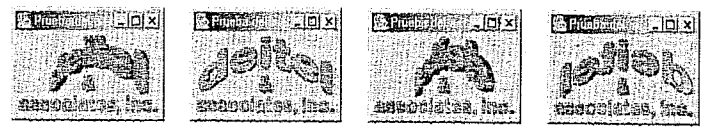


Figura 19.2 Animación de una serie de imágenes. (Parte 3 de 3.)

La clase **AnimadorLogo** mantiene un arreglo de objetos **ImageIcon** que se cargan en el constructor (líneas 21 a 33). En las líneas 26 a 32 se crea cada uno de los objetos **ImageIcon** y se cargan las 30 imágenes de la animación. El argumento del constructor utiliza la concatenación de cadenas para construir el nombre del archivo a partir de las piezas **"imagenes/"**, **NOMBRE_IMAGEN**, **cuenta** y **".gif"**. Cada una de las imágenes de la animación se encuentra en un archivo llamado **deitel#.gif**, en donde # es un valor en el rango de 0 a 29, el cual se especifica mediante la variable de control de ciclo **cuenta**. En las líneas 31 y 32

http://librosysolucionarios.net

se determinan la anchura y altura de la animación, con base en el tamaño de la primera imagen del arreglo `imagenes`. Supongamos que todas las imágenes tienen la misma anchura y altura.

Una vez que el constructor de `AnimadorLogo` carga las imágenes, el método `main` establece la ventana en la que va a aparecer la animación (líneas 89 a 96), y en la línea 97 se hace una llamada al método `iniciarAnimacion` de `AnimadorLogo` (declarado en las líneas 54 a 64) para iniciar la animación. La animación es controlada por una instancia de la clase `Timer` (del paquete `javax.swing`). Un objeto `Timer` genera eventos `ActionEvent` en un intervalo fijo en milisegundos (que generalmente se especifica como argumento para el constructor de `Timer`) y notifica a todos sus objetos `ActionListener` que ocurrió el evento. En las líneas 56 a 60 se determina si la referencia `Timer` llamada `temporizadorAnimacion` es `null`. De ser así, en la línea 57 se establece `imagenActual` en 0, lo cual indica que la animación debe empezar con la imagen que se encuentra en el primer elemento del arreglo `imagenes`. En la línea 58 se asigna un nuevo objeto `Timer` a `temporizadorAnimacion`. El constructor de `Timer` recibe dos argumentos: el retraso en milisegundos (`retrasoAnimacion` es 50, según lo especificado en la línea 14) y el objeto `ActionListener` que responderá a los objetos `ActionEvent` de `Timer`. La clase `AnimadorLogo` implementa a `ActionListener`, por lo que en la línea 58 se especifica a `this` como el componente de escucha. En la línea 59 se inicia el objeto `Timer`. Una vez iniciado, `temporizadorAnimacion` generará un evento `ActionEvent` cada 50 milisegundos. En las líneas 62 y 63 se habilita el programa para reiniciar una animación que se haya detenido previamente. Por ejemplo, para hacer a una animación "amigable para el navegador" en un applet, la animación debe detenerse cuando el usuario cambie de páginas Web. Si el usuario regresa a la página Web con la animación, el método `iniciarAnimacion` puede usarse para reiniciar la animación. La condición `if` de la línea 62 utiliza el método `isRunning` de `Timer` para determinar si el objeto `Timer` está funcionando (es decir, generando eventos). Si no está funcionando, en la línea 63 se hace una llamada al método `restart` de `Timer` para indicar que el objeto `Timer` debe empezar a generar eventos otra vez.

En respuesta a cada evento de `Timer` en este ejemplo, el programa llama al método `actionPerformed` (líneas 48 a 51). En la línea 50 se hace una llamada al método `repaint` de `AnimadorLogo` para programar una llamada al método `paintComponent` de `AnimadorLogo` (líneas 36 a 45). Recuerde que cualquier subclase de `JComponent` que realice dibujos, deberá hacerlo en su método `paintComponent`. Recuerde que en el capítulo 14 vimos que la primera instrucción en cualquier método `paintComponent` debe ser una llamada al método `paintComponent` de la superclase, para asegurar que los componentes de Swing se muestren correctamente en pantalla.

En la línea 40 se pinta el objeto `ImageIcon` almacenado en el elemento `imagenActual` del arreglo. En las líneas 43 y 44 se determina si el objeto `temporizadorAnimacion` está funcionando y, de ser así, se prepara la siguiente imagen a mostrar en pantalla, incrementando el valor de `imagenActual` en 1. Observe el cálculo del residuo para asegurar que el valor de `imagenActual` sea 0 cuando se incremente más allá de 29 (el índice del último elemento en el arreglo). La instrucción `if` asegura que, si se hace una llamada a `paintComponent` mientras el objeto `Timer` está detenido, se mostrará la misma imagen. Esto podría ser útil si se proporciona una GUI que permita al usuario iniciar y detener la animación. Por ejemplo, si la animación se detiene y el usuario la cubre con otra ventana, y después la descubre, se realizará una llamada al método `paintComponent`. En este caso, no queremos que la animación muestre la siguiente imagen.

El método `detenerAnimacion` (líneas 67 a 70) detiene la animación, llamando al método `stop` de `Timer` para indicar que el objeto `Timer` debe dejar de generar eventos. Esto evita que `actionPerformed` llame a `repaint` para iniciar el proceso de pintar la siguiente imagen en el arreglo.

Observación de ingeniería de software 19.1



Al crear una animación para utilizarla en un applet, debe proporcionar un mecanismo para deshabilitar la animación cuando el usuario navega en una nueva página Web, distinta de la página en la que reside la aplicación.

Los métodos `getMinimumSize` (líneas 73 a 76) y `getPreferredSize` (líneas 79 a 82) sobrescriben a los métodos correspondientes heredados de la clase `Component`, y permiten a los administradores de esquemas determinar el tamaño apropiado de un objeto `AnimadorLogo` en un esquema. En este ejemplo, las imágenes tienen 160 píxeles de anchura y 80 píxeles de largo, por lo que el método `getPreferredSize` devuelve un objeto `Dimension` que contiene los números 160 y 80 (determinados en las líneas 31 y 32). El método `getMinimumSize` simplemente llama a `getPreferredSize` (una práctica común de programa-

ción) para indicar que los tamaños mínimo y preferido son iguales. Algunos administradores de esquemas ignoran las dimensiones especificadas por estos métodos. Por ejemplo, las regiones `NORTH` y `SOUTH` de un objeto `BorderLayout` utilizan solamente la altura preferida del componente.

19.4 Mapas de imágenes

Los *mapas de imágenes* son una técnica común utilizada para crear páginas Web interactivas. Un mapa de imágenes es una imagen con *áreas activas* en las que el usuario puede hacer clic para realizar una tarea, como cargar una página Web distinta en un navegador. Cuando el usuario posiciona el puntero del ratón sobre un área activa, generalmente aparece un mensaje descriptivo en el área de estado del navegador, o en un cuadro de información sobre herramientas.

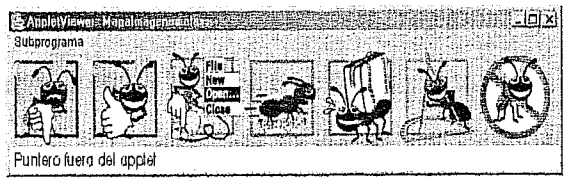
En la figura 19.3 se carga una imagen que contiene varios de los iconos de tips comunes que se utilizan en este libro. El programa permite al usuario posicionar el puntero del ratón sobre un icono y mostrar un mensaje descriptivo asociado con el icono. El manejador de eventos `mouseMoved` (líneas 37 a 41) toma las coordenadas del ratón y las pasa al método `traducirPosicion` (líneas 59 a 70). El método `traducirPosicion` evalúa las coordenadas para determinar sobre cuál icono se posicionó el ratón cuando ocurrió el evento `mouseMoved`; después el método devuelve un mensaje indicando lo que el icono representa. Este mensaje se muestra en la barra de estado del contenedor de applet.

```
1  // Fig. 19.3: MapaImagenes.java
2  // Demostración de un mapa de imágenes.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class MapaImagenes extends JApplet {
8     private ImageIcon imagenMapa;
9
10    private static final String leyendas[] = { "Error común de programación",
11       "Buena práctica de programación", "Observación de apariencia visual",
12       "Tip de rendimiento", "Tip de portabilidad",
13       "Observación de ingeniería de software", "Tip para prevenir errores" };
14
15    // establecer componentes de escucha del ratón
16    public void init()
17    {
18       addMouseListener(
19
20          new MouseAdapter() { // clase interna anónima
21
22             // indicar cuando el puntero del ratón salga del área del applet
23             public void mouseExited( MouseEvent evento )
24             {
25                showStatus( "Puntero fuera del applet" );
26             }
27
28          } // fin de la clase interna anónima
29
30       ); // fin de la llamada a addMouseListener
31
32       addMouseMotionListener(
33
34          new MouseMotionAdapter() { // clase interna anónima
```

Figura 19.3 Mapa de imágenes. (Parte 1 de 3.)

http://librosysolucionarios.net

```
35
36             // determinar el icono sobre el cual aparecerá el ratón
37             public void mouseMoved( MouseEvent evento )
38             {
39                showStatus( traducirPosicion(
40                   evento.getX(), evento.getY() ) );
41             }
42
43          } // fin de la clase interna anónima
44
45       ); // fin de la llamada a addMouseMotionListener
46
47       imagenMapa = new ImageIcon( "iconos.png" );  // obtener la imagen
48
49    }  // fin del método init
50
51    // mostrar imagenMapa
52    public void paint( Graphics g )
53    {
54       super.paint( g );
55       imagenMapa.paintIcon( this, g, 0, 0 );
56    }
57
58    // devolver leyenda del tip correspondiente, con base en las coordenadas del ratón
59    public String traducirPosicion( int x, int y )
60    {
61       // si las coordenadas están fuera de la imagen, regresar inmediatamente
62       if ( x >= imagenMapa.getIconWidth() || y >= imagenMapa.getIconHeight() )
63          return "";
64
65       // determinar número de icono (0 - 6)
66       int anchuraIcono = imagenMapa.getIconWidth() / 7;
67       int numeroIcono = x / anchuraIcono;
68
69       return leyendas[ numeroIcono ]; // devolver la leyenda del icono apropiado
70    }
71
72 } // fin de la clase MapaImagenes
```



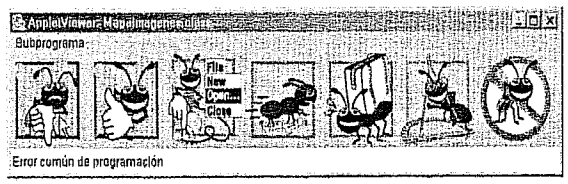


Figura 19.3 Mapa de imágenes. (Parte 2 de 3.)

http://librosysolucionarios.net

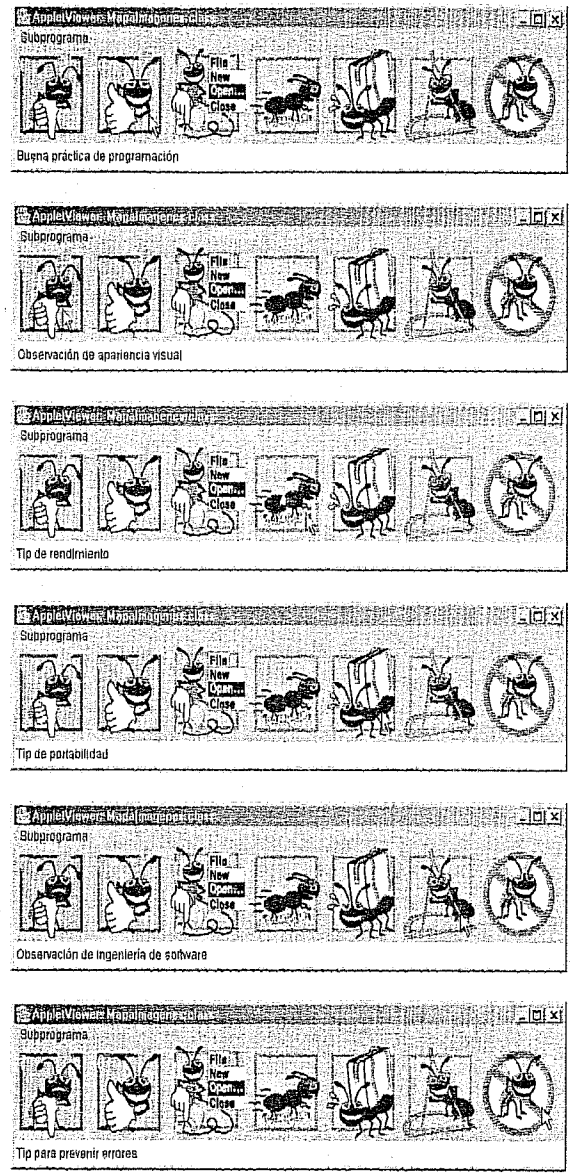


Figura 19.3 Mapa de imágenes. (Parte 3 de 3.)

Si hace clic en el applet de la figura 19.3, no se provocará ninguna acción. En el capítulo 18, Redes, hablamos sobre las técnicas requeridas para cargar otra página Web en un navegador mediante objetos `URL` y la interfaz `AppletContext`. Utilizando esas técnicas, este applet podría asociar cada icono con un objeto `URL` que el navegador mostraría cuando el usuario hiciera clic en el icono.

19.5 Carga y reproducción de clips de audio

Los programas de Java pueden manipular y reproducir *clips de audio*. Es fácil para los usuarios capturar sus propios clips de audio, y hay muchos clips disponibles en productos de software y a través de Internet. Su sistema necesita estar equipado con hardware de audio (bocinas y una tarjeta de sonido) para poder reproducir los clips de audio.

Java proporciona varios mecanismos para reproducir sonidos en un applet. Los dos métodos más simples son el método **`play`** de **`Applet`** y el método **`play`** de la interfaz **`AudioClip`**. Hay varias capacidades de audio adicionales disponibles en el Marco de trabajo de medios de Java (Java Media Framework) y en las APIs de sonido de Java (Java Sound APIs). Si desea reproducir un sonido una vez en un programa, el método **`play`** de **`Applet`** carga el sonido y lo reproduce una vez; el sonido se marca para la recolección de basura una vez que se reproduce. El método **`play`** de **`Applet`** tiene dos formas:

```
public void play( URL ubicacion, String nombreArchivoSonido );
public void play( URL urlSonido );
```

La primera versión carga el clip de audio almacenado en el archivo **`nombreArchivoSonido`** que se encuentra en **`ubicacion`** y reproduce el sonido. El primer argumento es normalmente una llamada al método **`getDocumentBase`** o ***`getCodeBase`*** del applet. El método **`getDocumentBase`** devuelve la ubicación del archivo HTML que cargó el applet. (Si el applet se encuentra en un paquete, el método devuelve la ubicación del paquete o archivo JAR que contiene a ese paquete.) El método **`getCodeBase`** indica la ubicación del archivo **`.class`** del applet. La segunda versión del método **`play`** toma un objeto **`URL`** que contiene la ubicación y el nombre de archivo del audio clip. La instrucción:

```
play( getDocumentBase(), "hi.au" );
```

carga el clip de audio que se encuentra en el archivo **`hi.au`** y reproduce el clip una vez.

El *motor de sonido* que reproduce los clips de audio soporta varios formatos de archivo de audio, incluyendo los formatos con extensión **`.au`** *(formato de archivo de audio de Sun)*, **`.wav`** *(formato de archivo de onda de Windows)*, **`.aif`** o **`.aiff`** *(formato de archivo AIFF de Macintosh)* y **`.mid`** o **`.rmi`** *(MIDI, Interfaz digital para instrumentos musicales)*. El JMF (Marco de medios de Java) y las APIs de sonido de Java soportan formatos adicionales.

El programa de la figura 19.4 demuestra cómo cargar y reproducir un objeto **`AudioClip`** (paquete **`java.applet`**). Esta técnica es más flexible que el método **`play`** de **`Applet`**. Un applet puede utilizar un objeto **`AudioClip`** para almacenar audio que se use en repetidas ocasiones a lo largo de la ejecución de un programa. El método ***`getAudioClip`*** de **`Applet`** tiene dos formas que toman los mismos argumentos que el método **`play`** que se describió anteriormente. El método **`getAudioClip`** devuelve una referencia a un objeto **`AudioClip`**. Un objeto **`AudioClip`** tiene tres métodos: ***`play`***, ***`loop`*** y ***`stop`***. El método **`play`** reproduce el audio una sola vez. El método **`loop`** reproduce continuamente el clip de audio en segundo plano. El método **`stop`** termina el clip de audio que se esté reproduciendo en ese momento. En el programa, cada uno de estos métodos se asocia con un botón en el applet.

```
1  // Fig. 19.4: CargarAudioYReproducir.java
2  // Cargar un clip de audio y reproducirlo.
3
4  import java.applet.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class CargarAudioYReproducir extends JApplet {
10    private AudioClip sonido1, sonido2, sonidoActual;
11    private JButton reproducirSonido, sonidoContinuo, detenerSonido;
12    private JComboBox seleccionarSonido;
13
```

Figura 19.4 Cómo cargar y reproducir un objeto **`AudioClip`**. (Parte 1 de 3.)

http://librosysolucionarios.net


```
14    // cargar la imagen cuando el applet empiece a ejecutarse
15    public void init()
16    {
17       Container contenedor = getContentPane();
18       contenedor.setLayout( new FlowLayout() );
19
20       String opciones[] = { "Welcome", "Hi" };
21       seleccionarSonido = new JComboBox( opciones );
22
23       seleccionarSonido.addItemListener(
24
25          new ItemListener() {
26
27             // detener sonido y cambiarlo por el que seleccionó el usuario
28             public void itemStateChanged( ItemEvent e )
29             {
30                sonidoActual.stop();
31
32                sonidoActual =
33                   seleccionarSonido.getSelectedIndex() == 0 ? sonido1 : sonido2;
34             }
35
36          } // fin de la clase interna anónima
37
38       ); // fin de la llamada al método addItemListener
39
40       contenedor.add( seleccionarSonido );
41
42       // establecer botones y manejador de eventos de botón
43       ButtonHandler manejador = new ButtonHandler();
44
45       reproducirSonido = new JButton( "Reproducir" );
46       reproducirSonido.addActionListener( manejador );
47       contenedor.add( reproducirSonido );
48
49       sonidoContinuo = new JButton( "Continuo" );
50       sonidoContinuo.addActionListener( manejador );
51       contenedor.add( sonidoContinuo );
52
53       detenerSonido = new JButton( "Detener" );
54       detenerSonido.addActionListener( manejador );
55       contenedor.add( detenerSonido );
56
57       // cargar sonidos y establecer sonidoActual
58       sonido1 = getAudioClip( getDocumentBase(), "welcome.wav" );
59       sonido2 = getAudioClip( getDocumentBase(), "hi.au" );
60       sonidoActual = sonido1;
61
62    } // fin del método init
63
64    // detener el sonido cuando el usuario cambie de página Web
65    public void stop()
66    {
67       sonidoActual.stop();
68    }
```

Figura 19.4 Cómo cargar y reproducir un objeto **`AudioClip`**. (Parte 2 de 3.)

```
69
70    // clase interna privada para manejar eventos de botón
71    private class ButtonHandler implements ActionListener {
72
73       // procesar eventos de reproducir, sonido continuo y detener
74       public void actionPerformed( ActionEvent eventoAccion )
75       {
76          if ( eventoAccion.getSource() == reproducirSonido )
77             sonidoActual.play();
78
79          else if ( eventoAccion.getSource() == sonidoContinuo )
80             sonidoActual.loop();
81
82          else if ( eventoAccion.getSource() == detenerSonido )
83             sonidoActual.stop();
84       }
85
86    } // fin de la clase ButtonHandler
87
88 } // fin de la clase CargarAudioYReproducir
```

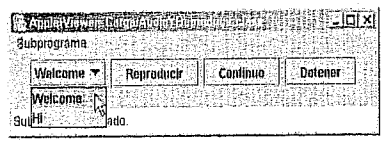


Figura 19.4 Cómo cargar y reproducir un objeto `AudioClip`. (Parte 3 de 3.)

Las líneas 58 y 59 en el método `init` del applet utilizan a `getAudioClip` para cargar dos archivos de audio: un archivo de onda de Windows (`welcome.wav`) y un archivo de audio de Sun (`hi.au`). El usuario puede seleccionar cuál clip de audio reproducir en el objeto `JComboBox` llamado `seleccionarSonido`. Observe que el método `stop` del applet se sobrescribe en las líneas 65 a 68. Cuando el usuario cambia de páginas Web, el contenedor de applet llama al método `stop` del applet. Esto permite al applet dejar de reproducir el clip de audio. De no ser así, el clip de audio continúa reproduciéndose en segundo plano; incluso aunque el applet no se muestre en el navegador. Esto no es necesariamente un problema, pero puede ser molesto para el usuario que el clip de audio se reproduzca en forma continua. El método `stop` se proporciona aquí como una conveniencia para el usuario.

Observación de apariencia visual 19.1



Al reproducir clips de audio en un applet o aplicación, debe proporcionar un mecanismo para que el usuario pueda deshabilitar el audio.

19.6 Recursos en Internet y World Wide Web

`www.nasa.gov/gallery/index.html`
La *galería multimedia de la NASA* contiene una amplia variedad de imágenes, clips de audio y de vídeo que usted puede descargar y utilizar para probar sus programas de multimedia en Java.

`sunsite.sut.ac.jp/multimed/`
La *Colección multimedia de Sunsite Japan* también proporciona una amplia variedad de imágenes, clips de audio y vídeo que usted puede descargar para fines educativos.

`www.anbg.gov.au/anbg/index.html`
El sitio Web *Jardines botánicos nacionales de Australia* proporciona vínculos a sonidos de muchos animales. Por ejemplo, puede probar el vínculo *Common Birds* que se encuentra dentro de la sección "Animals in the Botanic Gardens".

`www.thefreesite.com`
El sitio *TheFreeSite.com* tiene vínculos a sonidos y clips de arte gratuitos.

`www.soundcentral.com`
SoundCentral proporciona clips de audio en formatos WAV, AU, AIFF y MIDI.

`www.animationfactory.com`
El sitio *Animation Factory* proporciona miles de animaciones GIF gratuitas para uso personal.

`www.clipart.com`
ClipArt.com es un servicio basado en suscripciones, que ofrece imágenes y sonidos.

`www.pngart.com`
PNGART.com proporciona más de 50,000 imágenes gratuitas en formato PNG, en un esfuerzo por ayudar a que este reciente formato de imágenes obtenga popularidad.

`developer.java.sun.com/developer/techDocs/hi/repository`
El *Depósito de gráficos de apariencia visual de Java (Java Look-and-Feel Graphics Repository)* proporciona imágenes estándar para usarlas en una GUI de Swing.

Referencias para las APIs de multimedia de Java

`java.sun.com/products/java-media/jmf/`
Ésta es la página inicial de la *API JMF (Java Media Framework)*. Aquí puede descargar la implementación de Sun más reciente de JMF. Este sitio también contiene la documentación para la API de JMF.

`java.sun.com/products/java-media/sound/`
La página inicial de la *API de sonido de Java*. Esta API proporciona herramientas para reproducir y grabar audio.

`java.sun.com/products/java-media/3D/`
La página inicial de la *API Java 3D*. Esta API puede utilizarse para producir imágenes tridimensionales, típicas de los videojuegos de la actualidad.

`developer.java.sun.com/developer/onlineTraining/java3d/`
Este sitio proporciona un tutorial acerca de la API Java 3D.

`java.sun.com/products/java-media/jai/`
La página inicial de la *API para manejo avanzado de imágenes en Java (Java Advanced Imaging)*. Esta API proporciona herramientas para procesar imágenes, como mejorar el contraste, recortar, escalar y la deformación geométrica.

`java.sun.com/products/java-media/speech/`
Ésta es la página inicial de la *API de voz de Java (Java Speech)*. Esta API permite a los programas realizar la síntesis y el reconocimiento de voz.

`freetts.sourceforge.net/docs/index.php`
FreeTTS es una implementación de la API de voz de Java.

`java.sun.com/products/java-media/2D/`
Ésta es la página inicial de la *API Java 2D*. Esta API (presentada en el capítulo 12) proporciona herramientas para gráficos bidimensionales complejos.

`java.sun.com/j2se/1.4.1/docs/guide/imageio/`
Este sitio contiene una descripción general de la *API de E/S de imágenes de Java (Java Image I/O API)*, la cual permite a los programas definir procesos personalizados para cargar y guardar imágenes en formatos que no sean soportados todavía por las APIs de Java.

19.7 (Ejemplo práctico opcional) Acerca de los objetos: Animación y sonido en la vista

Este ejemplo práctico está enfocado principalmente en el modelo MVC para nuestra simulación de elevador. Ahora que hemos completado nuestro diseño del modelo nos concentraremos en la *vista*, que proporciona la presentación visual del modelo. En nuestro ejemplo práctico, la vista (encapsulada en la clase `VistaElevador`) es un objeto `JPanel` que contiene a otros objetos `JPanel` "hijos", cada uno de los cuales representa a un objeto único en el modelo MVC (por ejemplo, un objeto `Persona`, un `Boton`, el `Elevador`). La clase

http://librosysolucionarios.net

`VistaElevador` es la clase más extensa en el ejemplo práctico. En esta sección hablaremos sobre los gráficos y las clases de sonido utilizadas por la clase `VistaElevador`. Presentaremos y explicaremos el resto del código incluido en la vista, en el apéndice E.

En la sección 3.7 creamos un diagrama de clases para nuestro modelo, localizando los sustantivos y frases nominales del enunciado del problema de la sección 2.9. Ignoramos varios de estos sustantivos, ya que no estaban asociados con el modelo MVC. Ahora enlistaremos los sustantivos y frases nominales que se aplican al proceso de mostrar el modelo MVC en pantalla:

- mostrar
- audio
- música de elevador

El sustantivo "mostrar" corresponde a la vista (o presentación visual) del modelo MVC. Como se describió en la sección 13.17, la clase `VistaElevador` agrega varias clases que conforman la vista. El "audio" se refiere a los efectos de sonido que nuestra simulación genera cuando ocurren varias acciones; creamos la clase `EfectosDeSonido` para generar estos efectos de sonido. La frase "música de elevador" se refiere a la música que se reproduce cuando la `Persona` viaja en el `Elevador`; utilizaremos la clase `EfectosDeSonido` para reproducir esta música también.

La vista muestra objetos del modelo MVC. Creamos la clase `PanelImagen` para representar objetos estacionarios en el modelo, como el `ConductoElevador`. Creamos la clase `PanelMovil`, la cual extiende a `PanelImagen`, para representar objetos móviles tales como el `Elevador`. Por último, creamos la clase `PanelAnimado`, la cual extiende a `PanelMovil`, para representar objetos móviles cuyas imágenes correspondientes cambian continuamente, como es el caso de una `Persona` (utilizamos varios fotogramas de animación para mostrar a la `Persona` caminando y después oprimiendo un botón). Utilizando estas clases, presentamos el diagrama de clases de la vista para nuestra simulación, en la figura 19.5.

Las notas indican las funciones que desempeñan las clases en el sistema. De acuerdo con el diagrama de clases, la clase `VistaElevador` representa a la vista, las clases `PanelImagen`, `PanelMovil` y `PanelAnimado` se relacionan con los gráficos, y la clase `EfectosDeSonido` se relaciona con el audio. La clase `VistaElevador` contiene varias instancias de las clases `PanelImagen`, `PanelMovil` y `PanelAnimado`, junto con una instancia de la clase `EfectosDeSonido`. En el apéndice F, asociamos cada uno de los objetos en el modelo con su clase correspondiente en la vista.

En esta sección hablaremos sobre las clases `PanelImagen`, `PanelMovil` y `PanelAnimado` para explicar los gráficos y la animación. Después hablaremos sobre la clase `EfectosDeSonido` para explicar la funcionalidad de audio.

PanelImagen

La `VistaElevador` utiliza objetos de las subclases de `JPanel` para representar y mostrar cada uno de los objetos en el modelo (como el `Elevador`, una `Persona`, el `ConductoElevador`, etcétera). La clase `PanelImagen` (figura 19.6) es una subclase de `JPanel` capaz de mostrar una imagen en una posición dada en la pantalla. La `VistaElevador` utiliza objetos `PanelImagen` para representar objetos estacionarios en el modelo, como el `ConductoElevador` y los dos objetos `Piso`. La clase `PanelImagen` contiene un atributo entero (`ID`, en la línea 16) que declara un identificador único, el cual se utiliza para rastrear el objeto `PanelImagen` en la vista, en caso de ser necesario. Este rastreo es útil cuando existen varios objetos de la misma clase en el modelo, como varios objetos `Persona`. La clase `PanelImagen` contiene el objeto `Point2D.Double` llamado `posicion` (línea 19), el cual representa la posición en pantalla del objeto `PanelImagen`. Más adelante veremos que la clase `PanelMovil`, que extiende a `PanelImagen`, declara la velocidad mediante valores `double`. Convertimos las coordenadas de `posicion` en valores `int` para colocar el objeto `PanelImagen` en pantalla (Java representa las coordenadas en pantalla como valores `int`) en el método `establecerPosicion` (líneas 90 a 94). La clase `PanelImagen` también contiene un objeto `ImageIcon` llamado `iconoImagen` (línea 22); el método `paintComponent` (líneas 54 a 60) muestra a `iconoImagen` en pantalla. En las líneas 41 y 42 se inicializa `iconoImagen`, utilizando un parámetro `String` que contiene el nombre de la imagen. Por último, la clase `PanelImagen` contiene el objeto `SwingUtilities` `Panel` (línea 25), el cual almacena todos los objetos hijos de la clase `PanelImagen` (u objetos de una subclase de `PanelImagen`). Los objetos hijos se muestran arriba de su objeto `PanelImagen` padre; por ejemplo, una `Persona` viajando dentro del `Elevador`. El primer método `add` (líneas 63 a 67) anexa un objeto a `hijosPanel`. El segundo método `add` (líneas 70 a 74) inserta un objeto en `hijosPanel`, en un índice dado. El método `establecerIcono` (líneas 84 a 87) asigna a `iconoImagen` una nueva imagen. Los objetos de la clase `PanelAnimado` utilizan el método `establecerIcono` en forma repetida para cambiar la imagen mostrada, lo cual produce la animación para la vista; hablaremos posteriormente sobre la animación en esta sección.

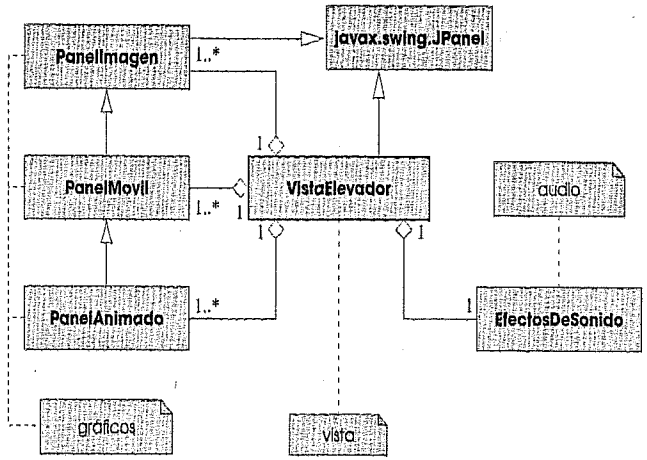


Figura 19.5 Diagrama de clases de la vista de la simulación del elevador.

```
1  // Fig. 19.6: PanelImagen.java
2  // Subclase de JPanel para posicionar y mostrar un objeto ImageIcon
3  package com.deitel.cpej5.elevador.vista;
4
5  // paquetes básicos de Java
6  import java.awt.*;
7  import java.awt.geom.*;
8  import java.util.*;
9
10 // paquetes de extensión de Java
11 import javax.swing.*;
12
13 public class PanelImagen extends JPanel {
14
15    // identificador
16    private int ID;
17
18    // posición en pantalla
19    private Point2D.Double posicion;
20
21    // iconoImagen a pintar en pantalla
```

Figura 19.6 La clase `PanelImagen` representa y muestra un objeto estacionario del modelo. (Parte 1 de 3.)

http://librosysolucionarios.net

```
22    private ImageIcon iconoImagen;
23
24    // almacena todos los hijos de PanelImagen
25    private Set hijosPanel;
26
27    // el constructor inicializa posición e imagen
28    public PanelImagen( int identificador, String nombreImagen )
29    {
30       super( null );      // especificar esquema nulo
31       setOpaque( false ); // hacer transparente
32
33       // establecer identificador único
34       ID = identificador;
35
36       // establecer posición
37       posicion = new Point2D.Double( 0, 0 );
38       establecerPosicion( 0, 0 );
39
40       // crear objeto ImageIcon con nombreImagen dado
41       iconoImagen = new ImageIcon(
42          getClass().getResource( nombreImagen ) );
43
44       Image imagen = iconoImagen.getImage();
45       setSize(
46          imagen.getWidth( this ), imagen.getHeight( this ) );
47
48       // crear objeto Set para almacenar hijos de Panel
49       hijosPanel = new HashSet();
50
51    } // fin del constructor de PanelImagen
52
53    // pintar Panel en la pantalla
54    public void paintComponent( Graphics g )
55    {
56       super.paintComponent( g );
57
58       // si la imagen está lista, pintarla en la pantalla
59       iconoImagen.paintIcon( this, g, 0, 0 );
60    }
61
62    // agregar hijo de PanelImagen a PanelImagen
63    public void add( PanelImagen panel )
64    {
65       hijosPanel.add( panel );
66       super.add( panel );
67    }
68
69    // agregar hijo de PanelImagen a PanelImagen en el índice dado
70    public void add( PanelImagen panel, int indice )
71    {
72       hijosPanel.add( panel );
73       super.add( panel, indice );
74    }
75
```

Figura 19.6 La clase **PanelImagen** representa y muestra un objeto estacionario del modelo. (Parte 2 de 3.)


```
76    // eliminar hijo PanelImagen de PanelImagen
77    public void remove( PanelImagen panel )
78    {
79       hijosPanel.remove( panel );
80       super.remove( panel );
81    }
82
83    // establece el objeto ImageIcon actual que se va a mostrar
84    public void establecerIcono( ImageIcon icono )
85    {
86       iconoImagen = icono;
87    }
88
89    // establecer posición en pantalla
90    public void establecerPosicion( double x, double y )
91    {
92       posicion.setLocation( x, y );
93       establecerPosicion( ( int ) x,  ( int ) y );
94    }
95
96    // devolver identificador PanelImagen
97    public int obtenerID()
98    {
99       return ID;
100   }
101
102   // obtener posición de objeto PanelImagen
103   public Point2D.Double obtenerPosicion()
104   {
105      return posicion;
106   }
107
108   // obtener iconoImagen
109   public ImageIcon obtenerIconoImagen()
110   {
111      return iconoImagen;
112   }
113
114   // obtener objeto Set de hijos PanelImagen
115   public Set obtenerHijos()
116   {
117      return hijosPanel;
118   }
119 }
```

Figura 19.6 La clase **PanelImagen** representa y muestra un objeto estacionario del modelo. (Parte 3 de 3.)

PanelMovil

La clase **PanelMovil** (figura 19.7) es una subclase de **PanelImagen** capaz de cambiar su posición en pantalla de acuerdo con sus valores **velocidadX** y **velocidadY** (líneas 20 y 21). El objeto **VistaElevador** utiliza objetos **PanelMovil** para representar objetos móviles del modelo, como el **Elevador**.

http://librosysolucionarios.net

```
1  // Fig. 19.7: PanelMovil.java
2  // Subclase de JPanel con herramientas para mover objetos en pantalla
3  package com.deitel.cpej5.elevador.vista;
4
5  // paquetes básicos de Java
6  import java.awt.*;
7  import java.awt.geom.*;
8  import java.util.*;
9
10 // paquetes de extensión de Java
11 import javax.swing.*;
12
13 public class PanelMovil extends PanelImagen {
14
15    // ¿debe PanelMovil cambiar de posición?
16    private boolean movimiento;
17
18    // número de píxeles que se mueve PanelMovil en valores de x y y
19    // por el número de  milisegundos indicado en retrasoAnimacion
20    private double velocidadX;
21    private double velocidadY;
22
23    // el constructor inicializa la posición, velocidad e imagen
24    public PanelMovil( int identificador, String nombreImagen )
25    {
26       super( identificador, nombreImagen );
27
28       // establecer velocidad de PanelMovil
29       velocidadX = 0;
30       velocidadY = 0;
31
32    } // fin del constructor de PanelMovil
33
34    // actualizar posición de PanelMovil y cuadro de animación
35    public void animar()
36    {
37       // actualizar posición de acuerdo a la velocidad de PanelMovil
38       if ( seEstaMoviendo() ) {
39          double posicionXAntigua = obtenerPosicion().getX();
40          double posicionYAntigua = obtenerPosicion().getY();
41
42          establecerPosicion( posicionXAntigua + velocidadX,
43             posicionYAntigua + velocidadY );
44       }
45
46       // actualizar a todos los hijos de PanelMovil
47       Iterator iterador = obtenerHijos().iterator();
48
49       while ( iterador.hasNext() ) {
50          PanelMovil panel = ( PanelMovil ) iterador.next();
51          panel.animar();
52       }
53    } // fin del método animar
54
55    // ¿se está moviendo PanelMovil en la pantalla?
```

Figura 19.7 La clase **PanelMovil** representa y muestra un objeto móvil del modelo. (Parte 1 de 2.)


```
56    public boolean seEstaMoviendo()
57    {
58       return movimiento;
59    }
60
61    // establecer PanelMovil para que se mueva en la pantalla
62    public void establecerMovimiento( boolean mover )
63    {
64       movimiento = mover;
65    }
66
67    // establecer velocidad x y y de PanelMovil
68    public void establecerVelocidad( double x, double y )
69    {
70       velocidadX = x;
71       velocidadY = y;
72    }
73
74    // devolver velocidad x de PanelMovil
75    public double obtenerVelocidadX()
76    {
77       return velocidadX;
78    }
79
80    // devolver velocidad y de PanelMovil
81    public double obtenerVelocidadY()
82    {
83       return velocidadY;
84    }
85 }
```

Figura 19.7 La clase **PanelMovil** representa y muestra un objeto móvil del modelo. (Parte 2 de 2.)

El método **animar** (líneas 35 a 53) mueve el **PanelMovil** de acuerdo con los valores actuales de los campos **velocidadX** y **velocidadY**. Si la variable **boolean** llamada **movimiento** (línea 16) es **true**, en las líneas 38 a 44 se utilizan los campos **velocidadX** y **velocidadY** para determinar la siguiente ubicación para el **PanelMovil**. En las líneas 47 a 52 se repite el proceso para cualquier hijo. En nuestra simulación, **VistaElevador** invoca al método **animar** y al método **paintComponent** de la clase **PanelImagen** cada 50 milisegundos. Estas rápidas llamadas sucesivas mueven el objeto **PanelMovil**.

PanelAnimado

La clase **PanelAnimado** (figura 19.8), que extiende a la clase **PanelMovil**, representa a un objeto animado del modelo (es decir, los objetos móviles cuyas imágenes correspondientes cambian continuamente), como una **Persona**. La **VistaElevador** anima un objeto **PanelAnimado**, cambiando el mensaje asociado con **iconoImagen**.

```
1  // Fig. 19.8: PanelAnimado.java
2  // Subclase de PanelMovil con herramientas para animación
3  package com.deitel.cpej5.elevador.vista;
4
5  // paquetes básicos de Java
6  import java.awt.*;
```

Figura 19.8 La clase **PanelAnimado** representa y muestra a un objeto animado del modelo. (Parte 1 de 4.)

http://librosysolucionarios.net

```
7  import java.util.*;
8
9  // paquetes de extensión de Java
10 import javax.swing.*;
11
12 public class PanelAnimado extends PanelMovil {
13
14    // ¿debe el objeto ImageIcon recorrer los cuadros?
15    private boolean animacion;
16
17    // velocidad de cuadro (es decir, la velocidad a la que se avanza al siguiente cuadro)
18    private int velocidadAnimacion;
19    private int contadorVelocidadAnimacion;
20    private boolean recorrerHaciaDelante = true;
21
22    // objetos ImageIcon individuales utilizados para los cuadros de animación
23    private ImageIcon iconosImagenes[];
24
25    // almacenamiento para todas las secuencias de cuadros
26    private java.util.List secuenciasCuadros;
27    private int animacionActual;
28
29    // ¿debe volver a empezar (continuar) la animación al final del ciclo?
30    private boolean continuar;
31
32    // ¿debe la animación mostrar el último cuadro al final de la animación?
33    private boolean mostrarUltimoCuadro;
34
35    // ayuda a determinar el siguiente cuadro a mostrar
36    private int contadorCuadroActual;
37
38    // el constructor toma un arreglo de nombres de archivos y la posición en pantalla
39    public PanelAnimado( int identificador, String nombreImagen[] )
40    {
41       super( identificador, nombreImagen[0] );
42
43       // crea objetos ImageIcon a partir del arreglo de cadenas nombreImagen
44       iconosImagenes = new ImageIcon[ nombreImagen.length ];
45
46       for ( int i = 0; i < iconosImagenes.length; i++ ) {
47          iconosImagenes[i] = new ImageIcon(
48             getClass().getResource( nombreImagen[i] ) );
49       }
50
51       secuenciasCuadros = new ArrayList();
52
53    } // fin del constructor de PanelAnimado
54
55    // actualizar posición del icono y cuadro de animación
56    public void animar()
57    {
58       super.animar();
59
60       // reproducir siguiente cuadro de animación si contador > velocidad de animación
```

Figura 19.8 La clase **PanelAnimado** representa y muestra a un objeto animado del modelo. (Parte 2 de 4.)


```
61       if ( secuenciasCuadros != null && estaAnimando() ) {
62
63          if ( contadorVelocidadAnimacion > velocidadAnimacion ) {
64             contadorVelocidadAnimacion = 0;
65             determinarSiguienteCuadro();
66          }
67          else
68             contadorVelocidadAnimacion++;
69       }
70    } // fin del método animar
71
72    // determinar siguiente cuadro de animación
73    private void determinarSiguienteCuadro()
74    {
75       int secuenciaCuadros[] =
76          ( int[] ) secuenciasCuadros.get( animacionActual );
77
78       // si no hay más cuadros de animación, determinar cuadro final,
79       // a menos que se especifique continuar
80       if ( contadorCuadroActual >= secuenciaCuadros.length ) {
81          contadorCuadroActual = 0;
82
83          // si continuar es false, terminar la animación
84          if ( !debeContinuar() ) {
85
86             establecerAnimacion( false );
87
88             if ( esMostrarUltimoCuadro() )
89
90                // mostrar el último cuadro en la secuencia
91                contadorCuadroActual = secuenciaCuadros.length - 1;
92          }
93       }
94
95       // establecer cuadro actual de animación
96       establecerCuadroActual( secuenciaCuadros[ contadorCuadroActual ] );
97       contadorCuadroActual++;
98
99    } // fin del método determinarSiguienteCuadro
100
101   // agregar secuencia del cuadro (animación) al objeto ArrayList llamado
      secuenciasCuadros
102   public void agregarSecuenciaCuadro( int secuenciaCuadros[] )
103   {
104      secuenciasCuadros.add( secuenciaCuadros );
105   }
106
107   // preguntar si PanelAnimado está animando (recorriendo cuadros)
108   public boolean estaAnimando()
109   {
110      return animacion;
111   }
112
113   // establecer PanelAnimado para que anime
114   public void establecerAnimacion( boolean animar )
```

Figura 19.8 La clase **PanelAnimado** representa y muestra a un objeto animado del modelo. (Parte 3 de 4.)

http://librosysolucionarios.net

```
115    {
116       animacion = animar;
117    }
118
119    // establecer objeto ImageIcon actual
120    public void establecerCuadroActual( int cuadro )
121    {
122       establecerIcono( iconosImagenes[ cuadro ] );
123    }
124
125    // establecer velocidad de animación
126    public void establecerVelocidadAnimacion( int velocidad )
127    {
128       velocidadAnimacion = velocidad;
129    }
130
131    // obtener velocidad de animación
132    public int obtenerVelocidadAnimacion()
133    {
134       return velocidadAnimacion;
135    }
136
137    // establecer si la animación debe reproducirse en forma continua
138    public void establecerContinua( boolean continuarAnimacion  )
139    {
140       continuar = continuarAnimacion;
141    }
142
143    // ver si la animación debe continuar
144    public boolean debeContinuar()
145    {
146       return continuar;
147    }
148
149    // ver si se va a mostrar el último cuadro al final de la animación
150    private boolean esMostrarUltimoCuadro()
151    {
152       return mostrarUltimoCuadro;
153    }
154
155    // establecer si se va a mostrar el último cuadro al final de la animación
156    public void establecerMostrarUltimoCuadro( boolean mostrarCuadro )
157    {
158       mostrarUltimoCuadro = mostrarCuadro;
159    }
160
161    // empezar a reproducir la secuencia de animación del índice dado
162    public void reproducirAnimacion( int secuenciaCuadros )
163    {
164       animacionActual = secuenciaCuadros;
165       contadorCuadroActual = 0;
166       establecerAnimacion( true );
167    }
168 }
```

Figura 19.8 La clase **PanelAnimado** representa y muestra a un objeto animado del paquete. (Parte 4 de 4.)

http://librosysolucionarios.net

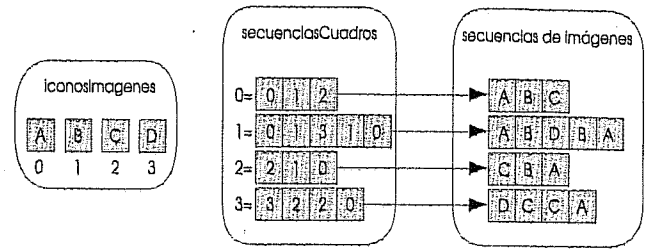


Figura 19.9 Relación entre el arreglo **iconosImagenes** y el objeto **List secuenciasCuadros**.

La clase **PanelAnimado** selecciona el objeto **ImageIcon** que se va a dibujar en la pantalla, de entre varios objetos **ImageIcon** almacenados en el arreglo **iconosImagenes** (línea 23). La clase **PanelAnimado** determina el objeto **ImageIcon** de acuerdo con una serie de referencias de *secuencia de cuadro*, almacenadas en el objeto **List** llamado **secuenciasCuadros** (línea 26). Una secuencia de cuadros es un arreglo de enteros que almacena la secuencia apropiada para mostrar los objetos **ImageIcon**; específicamente, cada entero representa el índice de un objeto **ImageIcon** en **iconosImagenes**. En la figura 19.9 se demuestra la relación entre **iconosImagenes** y **secuenciasCuadros**. (Éste no es un diagrama de UML.) Por ejemplo, el número de secuencia de cuadro:

```
2 = { 2 , 1 , 0 }
```

hace referencia a { **iconoImagen[2], iconoImagen[1], iconoImagen[0]** }, lo cual produce la secuencia de imágenes { **C , B , A** }. En la vista, cada imagen es un archivo **.png** único. El método **agregarSecuenciaCuadro** (líneas 102 a 105) agrega una secuencia de cuadro al objeto **List** llamado **secuenciasCuadros**. El método **reproducirAnimacion** (líneas 162 a 167) inicia la animación asociada con el parámetro **secuenciaCuadros**. Por ejemplo, suponga que hay un objeto **PanelAnimado** llamado **personaPanelAnimado** en la clase **VistaElevador**. El siguiente segmento de código:

```
panelAnimado.reproducirAnimacion( 1 );
```

generaría la secuencia de imágenes { **A , B , D , B , A** }, utilizando la figura 19.9 como una referencia.

El método **animar** (líneas 56 a 70) sobrescribe al método **animar** de la superclase **PanelMovil**. En las líneas 61 a 69 se determina el siguiente cuadro de animación, dependiendo del campo **velocidadAnimacion**, el cual es inversamente proporcional a la velocidad de animación; un valor mayor para **velocidadAnimacion** produce una velocidad de animación menor. Por ejemplo, si **velocidadAnimacion** es 5, **animar** avanza al siguiente cuadro de animación cada cinco veces que sea invocado. Utilizando esta lógica, la velocidad de animación se maximiza cuando **velocidadAnimacion** tiene un valor de 1, ya que el siguiente cuadro se determina cada vez que se ejecuta el método **animar**.

El método **animar** llama a **determinarSiguienteCuadro** (líneas 73 a 99) para determinar el siguiente cuadro (imagen) a mostrar; específicamente, llama al método **establecerCuadroActual** (líneas 120 a 123), el cual asigna a **iconoImagen** (la imagen que se muestra en ese momento) la imagen devuelta de la secuencia de cuadros actual. Las líneas 84 a 92 de **determinarSiguienteCuadro** se utilizan para reproducir la animación en forma continua. Si **continuar** es **false**, la animación termina después de una iteración. Se muestra el último cuadro en la secuencia si **mostrarUltimoCuadro** es **true**, y se muestra el primer cuadro en la secuencia si **mostrarUltimoCuadro** es **false**. En el apéndice F explicaremos con más detalle cómo **VistaElevador** utiliza a **mostrarUltimoCuadro** para los objetos **PanelAnimado** **Persona** y **Puerta**, para asegurar que la imagen se muestre correctamente. Si **continuar** es **true**, la animación se repite hasta que se detenga explícitamente.

Efectos de sonido

Ahora veremos cómo generar audio en nuestra simulación del elevador. La clase **EfectosDeSonido** (figura 19.10) manipula los archivos de audio (**.au**), de onda (**.wav**) y MIDI (**.mid**), que contienen sonidos como el timbre, los pasos de la persona y la música del elevador, en objetos **java.applet.AudioClip**.

En el apéndice I enlistamos todos los objetos **AudioClip** utilizados en nuestra simulación. El objeto **VistaElevador** reproducirá los objetos **AudioClip** para generar sonido. Todos los archivos de sonido se encuentran en la siguiente estructura de directorios:

```
com/deitel/cpej5/elevador/vista/sonidos
```

(es decir, en el directorio **sonidos**, en donde se encuentran ubicadas las clases para la vista en el sistema de archivos). En nuestra simulación, utilizamos sonidos y archivos **MIDI** que son proporcionados sin costo para descargarse del sitio Web de Microsoft:

```
msdn.microsoft.com/downloads/default.asp
```

La clase **EfectosDeSonido** contiene el método **obtenerClipAudio** (líneas 16 a 27), el cual utiliza el método **static newAudioClip** (de la clase **java.applet.Applet**) para devolver un objeto **AudioClip**, utilizando el parámetro **archivoSonido**. El método **establecerPrefijoRuta** (líneas 30 a 33) permite cambiar el directorio de un archivo de sonido (lo cual es útil si queremos repartir nuestros sonidos entre varios directorios).

```
1  // Fig. 19.10: EfectosDeSonido.java
2  // Devuelve objetos AudioClip
3  package com.deitel.cpej5.elevador.vista;
4
5  // paquetes básicos de Java
6  import java.applet.*;
7
8  public class EfectosDeSonido {
9
10    // ubicación de los archivos de sonido
11    private String prefijo = "";
12
13    public EfectosDeSonido() {}
14
15    // obtener objeto AudioClip asociado con archivoSonido
16    public AudioClip obtenerClipAudio( String archivoSonido )
17    {
18       try {
19          return Applet.newAudioClip( getClass().getResource(
20             prefijo + archivoSonido ) );
21       }
22
23       // devolver null si archivoSonido no existe
24       catch ( NullPointerException excepcionPunteroNulo ) {
25          return null;
26       }
27    }
28
29    // establecer el prefijo para la ubicación de archivoSonido
30    public void establecerPrefijoRuta( String cadena )
31    {
32       prefijo = cadena;
33    }
34 }
```

Figura 19.10 La clase **EfectosDeSonido** devuelve objetos **AudioClip**.

http://librosysolucionarios.net

Conclusión

Acaba usted de completar un considerable proceso de DOO (diseño orientado a objetos), creado para ayudarlo a prepararse para los retos que presentan los proyectos de "calidad industrial". Esperamos que las secciones opcionales "Acerca de los objetos" hayan sido informativas y útiles para usted, como suplementos para el material que presentamos en los capítulos. Además, esperamos que haya disfrutado la experiencia de diseñar el sistema del elevador utilizando UML. La industria de software a nivel mundial ha adoptado a UML como el estándar *de facto* para modelar software orientado a objetos.

Aunque hemos completado ya el proceso de diseño, apenas si hemos "raspado la superficie" del proceso de implementación. Le recomendamos que lea los apéndices D, E y F en el CD complementario, en los cuales se implementa el diseño completamente. Estos apéndices traducen los diagramas de UML en un programa completo en Java para la simulación del elevador. En estos apéndices presentamos todo el código que no vimos en las secciones "Acerca de los objetos", además de un "recorrido" completo a través de este código.

1. En el apéndice D se presentan los archivos de Java que implementan a los eventos y componentes de escucha.
2. En el apéndice E se presentan los archivos de Java que implementan el modelo.
3. En el apéndice F se presentan los archivos de Java que implementan la vista.

No presentamos mucho material nuevo ni diseño de UML en estos apéndices; simplemente sirven para implementar el diagrama basado en UML, que hemos presentado en capítulos anteriores, en un programa completamente funcional. Al estudiar la implementación en los apéndices usted podrá afinar las habilidades de programación que ha desarrollado a lo largo del libro, además de reforzar su comprensión acerca del proceso de diseño.

RESUMEN

- El método **getImage** de **Applet** carga un objeto **Image**.
- El método **getDocumentBase** de **Applet** devuelve la ubicación del archivo HTML del applet en Internet, como un objeto de la clase **URL** (paquete **java.net**).
- Java soporta varios formatos de imágenes, incluyendo GIF (Formato de intercambio de gráficos), JPEG (Grupo unido de expertos en fotografía) y PNG (Gráficos portables de red). Los nombres de archivo para cada uno de estos tipos terminan con **.gif**, **.jpg** (o **.jpeg**) y **.png**, respectivamente.
- La clase **ImageIcon** proporciona constructores que permiten a un objeto **ImageIcon** inicializarse con una imagen del equipo local, o con una imagen almacenada en un servidor Web en Internet.
- El método **drawImage** de **Graphics** recibe cuatro argumentos: una referencia al objeto **Image** en el que se almacena la imagen, las coordenadas *x* y *y* en donde debe mostrarse la imagen y una referencia a un objeto **ImageObserver**.
- Otra versión del método **drawImage** de **Graphics** muestra en pantalla una imagen *escalada*. Los argumentos cuarto y quinto especifican la anchura y altura de la imagen para mostrarla en pantalla.
- La interfaz **ImageObserver** es implementada por la clase **Component**. Los objetos **ImageObserver** son notificados para actualizar una imagen que se haya mostrado a medida que se carga el resto de la imagen.
- El método **paintIcon** de **ImageIcon** muestra la imagen del objeto **ImageIcon**. Este método requiere cuatro argumentos: una referencia al objeto **Component** en el que se va a mostrar la imagen, una referencia al objeto **Graphics** que se utilizará para desplegar la imagen, la coordenada *x* de la esquina superior izquierda de la imagen y la coordenada *y* de la esquina superior izquierda de la imagen.
- La clase **ImageIcon** proporciona el método **getImage**, el cual devuelve una referencia **Image** que el método **drawImage** de **Graphics** puede utilizar para mostrar una versión a escala de una imagen.
- Los objetos **Timer** generan eventos **ActionEvent** en intervalos fijos, en milisegundos. El constructor de **Timer** recibe dos argumentos: el retraso en milisegundos y el objeto **ActionListener**. El método **start** de **Timer** indica que el objeto **Timer** debe empezar a generar eventos. El método **stop** de **Timer** indica que el objeto **Timer** debe dejar de generar eventos. El método **restart** de **Timer** indica que el objeto **Timer** debe empezar a generar eventos otra vez.
- Una **imagen de imágenes** es una imagen que tiene áreas activas, en las que el usuario puede hacer clic para realizar una tarea, como cargar una página Web distinta en un navegador.

- El método `play` de `Applet` tiene dos formas:

```
public void play( URL ubicacion, String nombreArchivoSonido );
public void play( URL urlSonido );
```

  Una versión carga el clip de audio almacenado en el archivo `nombreArchivoSonido` desde la `ubicación` y reproduce el sonido; la otra versión toma un objeto `URL` que contiene la ubicación y el nombre de archivo del clip de audio.
- El método `getDocumentBase` de `Applet` indica la ubicación del archivo HTML que cargó el applet. El método `getCodeBase` indica en dónde se encuentra el archivo `.class` de un applet.
- El motor de sonido que reproduce clips de audio soporta varios formatos de archivo de audio, incluyendo los formatos con extensión `.au` (formato de archivo de audio de Sun), `.wav` (formato de archivo de onda de Windows), `.aif` o `.aiff` (formato de archivo AIFF de Macintosh) y `.mid` o `.rmi` (MIDI, Interfaz digital para instrumentos musicales). El Marco de medios de Java (JMF) soporta otros formatos adicionales.
- El método `getAudioClip` de `Applet` tiene dos formas que toman los mismos argumentos que el método `play`. El método `getAudioClip` devuelve una referencia a un objeto `AudioClip`. Un objeto `AudioClip` tiene tres métodos: `play`, `loop` y `stop`. El método `play` reproduce el audio una sola vez. El método `loop` reproduce continuamente el clip de audio. El método `stop` termina el clip de audio que se esté reproduciendo en ese momento.

TERMINOLOGÍA

`.aif`, extensión de nombre de archivo
`.aiff`, extensión de nombre de archivo
`.au`, extensión de nombre de archivo
`.gif`, extensión de nombre de archivo
`.jpeg`, extensión de nombre de archivo
`.jpg`, extensión de nombre de archivo
`.mid`, extensión de nombre de archivo
`.rmi`, extensión de nombre de archivo
`.wav`, extensión de nombre de archivo
altura de la imagen
anchura de una imagen
animación
animación de una serie de imágenes
clip de audio
cuadro de información
`drawImage`, método de `Graphics`
escalar una imagen
formato de archivo (`.au`)
Formato de intercambio de gráficos (GIF)
`getAudioClip`, método de `Applet`
`getCodeBase`, método de `Applet`
`getDocumentBase`, método de `Applet`
`getHeight`, método de `Component`
`getIconHeight`, método de `ImageIcon`
`getIconWidth`, método de `ImageIcon`
`getImage`, método de `Applet`
`getImage`, método de `ImageIcon`
`getWidth`, método de `Component`
Grupo unido de expertos en fotografía (JPEG)
`Image`, clase
`ImageIcon`, clase
imagen
`ImageObserver`, interfaz
Interfaz digital para instrumentos musicales (MIDI)
`loop`, método de la interfaz `AudioClip`
Macintosh AIFF, archivo (`.aif` o `.aiff`)
mapa de imágenes
motor de sonido
multimedia
Onda de Windows, archivo (.wav)
`paintIcon`, método de la clase `ImageIcon`
`play`, método de la clase `Applet`
`play`, método de la interfaz `AudioClip`
`restart`, método de la clase `Timer`
sonido
`start`, método de la clase `Timer`
`stop`, método de la clase `Timer`
`stop`, método de la interfaz `AudioClip`
`Timer`, clase
`update`, método de la clase `Component`

EJERCICIOS DE AUTOEVALUACIÓN

19.1 Complete las siguientes oraciones:
a) El método _____ de `Applet` carga una imagen en un applet.
b) El método _____ de `Applet` devuelve, como un objeto de la clase `URL`, la ubicación en Internet del archivo HTML que invocó al applet.
c) El método _____ de `Graphics` muestra una imagen en un applet.
d) Java proporciona dos mecanismos para reproducir sonidos en un applet: el método `play` de `Applet` y el método `play` de la interfaz _____.

http://librosysolucionarios.net

e) Un _____ es una imagen que tiene *áreas activas*, en las que el usuario puede hacer clic para realizar una tarea, como cargar una página Web distinta.
f) El método _____ de la clase `ImageIcon` muestra la imagen del objeto `ImageIcon`.
g) Java soporta varios formatos de imagen, incluyendo _____, _____ y _____.

19.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Un sonido será recolectado como basura tan pronto como haya dejado de reproducirse.
b) La clase `ImageIcon` proporciona constructores que permiten a un objeto `ImageIcon` ser inicializado con sólo una imagen proveniente del equipo local.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

19.1 a) `getImage`. b) `getDocumentBase`. c) `drawImage`. d) `AudioClip`. e) Mapa de imágenes. f) `paintIcon`. g) Formato de intercambio de gráficos. (GIF), Grupo unido de expertos en fotografía (JPEG), Gráficos portables de red (PNG).

19.2 a) Falso. El sonido será candidato para la recolección de basura (si no es referenciado por un objeto `AudioClip`) y se recolectará como basura cuando el recolector pueda ejecutarse. b) Falso. `ImageIcon` puede cargar imágenes de Internet también.

EJERCICIOS

19.3 Describa cómo hacer que una animación sea "amigable para el navegador".

19.4 Describa los métodos de Java para reproducir y manipular clips de audio.

19.5 Explique cómo se utilizan los mapas de imágenes. Enliste varios ejemplos en los que se utilicen mapas de imágenes.

19.6 *(Borrar una imagen al azar.)* Suponga que se muestra una imagen en un área rectangular de la pantalla. Una manera de borrar la imagen es simplemente asignar a cada píxel el mismo color inmediatamente, pero éste es un efecto visual torpe. Escriba un programa en Java para mostrar una imagen y luego borrarla, utilizando la generación de números aleatorios para seleccionar píxeles individuales y borrarlos. Una vez que se haya borrado la mayor parte de la imagen, borre el resto de los píxeles al mismo tiempo. Puede dibujar píxeles individuales como una línea que inicie y termine en las mismas coordenadas. Tal vez sea conveniente que pruebe distintas variantes de este problema. Por ejemplo, podría mostrar líneas o figuras al azar, para borrar regiones de la pantalla.

19.7 *(Texto intermitente.)* Cree un programa en Java para mostrar repetidamente texto intermitente en la pantalla. Para ello, alterne el texto con una imagen de fondo simple a color. Permita al usuario controlar la "velocidad de parpadeo" y el color o patrón de fondo.

19.8 *(Imagen intermitente.)* Cree un programa en Java para mostrar repetidamente una imagen intermitente en la pantalla. Para ello, alterne la imagen con una imagen de fondo simple a color.

19.9 *(Reloj digital.)* Implemente un programa para mostrar un reloj digital en la pantalla. Puede agregar opciones para escalar el reloj; mostrar el día, mes y año; emitir una alarma; reproducir ciertos clips de audio en tiempos designados, etcétera.

19.10 *(Atraer la atención hacia una imagen.)* Si desea enfatizar una imagen, puede colocar una fila de bombillas simuladas alrededor de su imagen. Puede dejar que las bombillas parpadeen al unísono, o puede dejar que se prendan y apaguen en secuencia, una después de otra.

19.11 *(Acercamiento/alejamiento de imagen.)* Cree un programa que le permita acercarse o alejarse de una imagen.

SECCIÓN ESPECIAL: PROYECTOS DE MULTIMEDIA RETADORES

Los ejercicios anteriores están adaptados al texto y diseñados para evaluar la comprensión del lector en cuanto a los conceptos fundamentales de multimedia. En esta sección incluimos una colección de proyectos avanzados de multimedia. El lector encontrará que estos problemas son retadores, pero interesantes. Los problemas varían considerablemente en cuanto al grado de dificultad. Algunos requieren de una o dos horas para escribir e implementar el programa correspondiente. Otros son útiles como tareas de laboratorio, que podrían requerir de dos o tres semanas de estudio e implementación. Algunos son proyectos finales retadores. (*Nota:* No se proporcionan las soluciones para estos ejercicios.)

19.12 *(Animación.)* Cree un programa de animación en Java, de propósito general. Su programa deberá permitir al usuario especificar la secuencia de cuadros a mostrar, la velocidad a la que se van a mostrar las imágenes, los clips de audio que deberán reproducirse mientras la animación se ejecuta, y así sucesivamente.

19.13 *(Quintillas.)* Modifique el programa para escribir quintillas del ejercicio 10.10, para que cante las quintillas que su programa cree.

19.14 *(Transición aleatoria entre imágenes.)* Este proyecto proporciona un agradable efecto visual. Si va a mostrar una imagen en cierta área de la pantalla y desea cambiar a otra imagen en la misma área de la pantalla, almacene la nueva imagen de pantalla en un búfer fuera de la pantalla, y copie al azar píxeles de la nueva imagen hacia el área que se va a mostrar en pantalla, cubriendo los píxeles anteriores en esas posiciones. Cuando se haya copiado la mayor parte de los píxeles, copie toda la nueva imagen en el área que se va a mostrar en pantalla, para asegurarse de estar mostrando la nueva imagen completa. Para implementar este programa, tal vez necesite usar las clases `PixelGrabber` y `MemoryImageSource` (consulte la documentación de las APIs de Java para obtener las descripciones de esas clases). Tal vez sea conveniente que pruebe distintas variantes de este problema. Por ejemplo, trate de seleccionar todos los píxeles en una línea recta o figura seleccionada al azar en la nueva imagen, y cubra esos píxeles por encima de las posiciones correspondientes de la imagen anterior.

19.15 *(Audio de fondo.)* Agregue audio de fondo a una de sus aplicaciones favoritas, utilizando el método `loop` de la clase `AudioClip` para reproducir el sonido en segundo plano mientras interactúa con su aplicación en forma normal.

19.16 *(Marquesina desplazable.)* Cree un programa en Java para desplazar caracteres punteados de derecha a izquierda (o de izquierda a derecha, si es apropiado para su lenguaje), a lo largo de un letrero tipo Marquesina. Como una opción, muestre el texto en un ciclo continuo de manera que el texto desaparezca en un extremo y reaparezca en el otro.

19.17 *(Marquesina con imagen desplazable.)* Cree un programa en Java para mostrar una imagen a lo largo de una pantalla tipo Marquesina.

19.18 *(Reloj análogo.)* Cree un programa en Java para mostrar un reloj análogo con manecillas para las horas, minutos y segundos que se desplacen apropiadamente, a medida que vaya cambiando la hora.

19.19 *(Audio dinámico y calidoscopio gráfico.)* Escriba un programa de calidoscopio que muestre gráficos reflejados para simular el popular juguete para niños. Incorpore efectos de audio que "reflejen" los gráficos de su programa que cambian en forma dinámica.

19.20 *(Generador automático de rompecabezas.)* Cree un generador y manipulador de rompecabezas en Java. El usuario especifica una imagen. Su programa carga y muestra la imagen. Su programa después divide la imagen en varias figuras seleccionadas al azar, y las revuelve. El usuario entonces utiliza el ratón para desplazar las piezas alrededor y tratar de resolver el rompecabezas. Agregue sonidos de audio apropiados a medida que se vayan moviendo las piezas y se inserten en el lugar correcto. Tal vez sea conveniente tener etiquetas en cada pieza y en el lugar en el que pertenece; después se pueden utilizar efectos de audio para ayudar al usuario a obtener las piezas en las posiciones correctas.

19.21 *(Programa para generar y recorrer laberintos.)* Desarrolle un programa basado en multimedia para generar laberintos y recorrerlos, con base en los programas de laberintos que escribió en los ejercicios 7.40 a 7.42. Deje que el usuario personalice el laberinto, especificando el número de filas y columnas junto con el nivel de dificultad. Haga que un ratón animado recorra el laberinto. Utilice audio para dramatizar el movimiento de su personaje ratón.

19.22 *(Bandido de un brazo.)* Desarrolle una simulación multimedia de una máquina tragamonedas, conocida como "bandido de un brazo". Debe tener tres ruedas giratorias. Coloque varias frutas y símbolos en cada rueda. Use la generación de números aleatorios para simular los giros de cada rueda y la acción de detenerse en un símbolo.

19.23 *(Carrera de caballos.)* Cree una simulación en Java de una carrera de caballos. Debe tener varios competidores. Use clips de audio para tener un anunciador. Reproduzca los clips de audio apropiados para indicar el estado correcto de cada uno de los competidores, a lo largo de la carrera. Use clips de audio para anunciar el resultado final. Sería conveniente que tratara de simular el tipo de juegos de carreras de caballos que se juegan en los carnavales. Los jugadores toman turnos en el ratón y tienen que realizar ciertas manipulaciones hábiles con el ratón para que sus caballos avancen.

19.24 *(Juego de tejo.)* Desarrolle una simulación basada en multimedia del juego de tejo. Use efectos de audio y visuales apropiados.

19.25 *(Juego de billar.)* Cree una simulación basada en multimedia del juego de billar. Cada jugador toma turnos utilizando el ratón para posicionar un taco y pegarle a la bola en el ángulo apropiado para tratar de hacer que las demás bolas caigan en las buchacas. Su programa deberá llevar la puntuación.

19.26 *(Artista.)* Diseñe un programa artístico en Java que proporcione a un artista una gran variedad de herramientas para dibujar, utilizar imágenes, animaciones, etcétera, para crear una muestra de arte dinámica con multimedia.

19.27 *(Diseñador de fuegos artificiales.)* Cree un programa en Java que podría ser utilizado para crear una muestra de fuegos artificiales. Cree una variedad de demostraciones de fuegos artificiales. Después controle la activación de los fuegos artificiales para obtener un efecto máximo.

19.28 *(Diseñador de pisos.)* Desarrolle un programa en Java que ayude a alguien a distribuir los muebles en su hogar. Agregue características que permitan a la persona lograr el mejor arreglo posible.

19.29 *(Crucigrama.)* Los crucigramas son uno de los pasatiempos más populares. Desarrolle un programa de crucigramas basado en multimedia. Su programa debe permitir al jugador colocar y borrar palabras fácilmente. Enlace su programa con un diccionario computarizado extenso. Su programa deberá también tener la capacidad de sugerir palabras en las que se hayan llenado algunas letras. Proporcione otras características que faciliten el trabajo de los entusiastas de crucigramas.

19.30 *(Acertijo del 15.)* Escriba un programa en Java basado en multimedia que permita al usuario jugar al 15. Este juego se lleva a cabo en un tablero de 4 por 4, para un total de 16 posiciones. Una de las posiciones está vacía. Las demás posiciones están ocupadas por 15 mosaicos, numerados del 1 al 15. Cualquier mosaico enseguida de la posición actual vacía puede moverse a esa posición, haciendo clic en el mosaico. Su programa deberá crear el tablero con los mosaicos desordenados. El objetivo es ordenar los mosaicos en forma secuencial, fila por fila.

19.31 *(Tiempo de reacción/Probador de precisión de reacción.)* Cree un programa en Java que mueva una figura creada al azar alrededor de la pantalla. El usuario desplazará el ratón para atrapar y hacer clic en la figura. Puede variarse la velocidad y el tamaño de la figura. Su programa deberá llevar las estadísticas acerca de cuánto tiempo le lleva al usuario atrapar una figura de un tamaño dado. Probablemente al usuario le sea más difícil atrapar figuras que sean más pequeñas y se muevan más rápido.

19.32 *(Calendario/archivo de recordatorios.)* Utilizando audio e imágenes, cree un calendario de propósito general y un archivo de "recordatorios". Por ejemplo, el programa deberá cantar "Feliz cumpleaños" cuando lo utilice en su cumpleaños. Haga que el programa muestre imágenes y reproduzca clips de audio asociados con eventos importantes. Además, haga que el programa le recuerde de antemano estos eventos importantes. Por ejemplo, sería bueno hacer que el programa le avisara con una semana de anticipación, para que pudiera recoger una tarjeta de felicitación apropiada para esa persona especial.

19.33 *(Imágenes giratorias.)* Cree un programa en Java que le permita girar una imagen un cierto número de grados (hasta un máximo de 360 grados). El programa deberá permitirle a usted especificar que desea girar la imagen en forma continua. El programa deberá permitirle ajustar la velocidad de giro en forma dinámica.

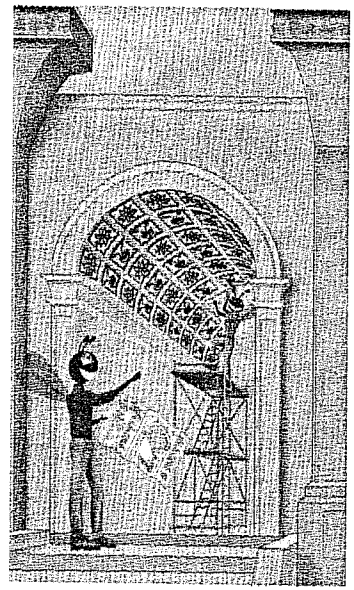
19.34 *(Colorear imágenes y fotografías en blanco y negro.)* Cree un programa en Java que le permita pintar de colores una fotografía en blanco y negro. Proporcione una paleta para seleccionar colores. Su programa deberá permitirle aplicar distintos colores en distintas regiones de la imagen.

19.35 *(Simulador Simpletron basado en multimedia.)* Modifique el simulador Simpletron que desarrolló en los ejercicios de capítulos anteriores, para incluir características de multimedia. Agregue sonidos tipo computadora para indicar que el Simpletron está ejecutando instrucciones. Agregue un sonido de vidrio quebrándose cuando ocurra un error fatal. Use luces intermitentes para indicar cuáles celdas de memoria o cuáles registros se están manipulando en ese momento. Use otras técnicas de multimedia según sea apropiado, para hacer que su simulador Simpletron sea más valioso para sus usuarios como una herramienta educativa.

http://librosysolucionarios.net

20

Estructuras de datos



Objetivos

- Aprender a formar estructuras de datos enlazadas utilizando referencias, clases autorreferenciadas y recursividad.
- Aprender a crear y manipular estructuras dinámicas de datos como listas enlazadas, colas, pilas y árboles binarios.
- Comprender varias aplicaciones importantes de las estructuras de datos enlazadas.
- Aprender a crear estructuras de datos reutilizables con clases, herencia y composición.

De muchas cosas a las que estoy atado, no he podido liberarme;
Y muchas de las que me liberé, han vuelto a mí.
Lee Wilson Dodd

'¿Podría caminar un poco más rápido?' dijo una merluza a un caracol,
'Hay una marsopa acercándose mucho a nosotros y está pisándome la cola.'
Lewis Carroll

Siempre hay lugar en la cima.
Daniel Webster

Empujen; sigan moviéndose.
Thomas Morton

Creo que nunca veré
un poema tan encantador como un árbol.
Joyce Kilmer

http://librosysolucionarios.net

Plan general

20.1 Introducción
20.2 Clases autorreferenciadas
20.3 Asignación dinámica de memoria
20.4 Listas enlazadas
20.5 Pilas
20.6 Colas
20.7 Árboles

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Sección especial: Construya su propio compilador

20.1 Introducción

Hemos estudiado *estructuras de datos* de tamaño fijo como los arreglos unidimensionales y multidimensionales. En este capítulo presentaremos las *estructuras de datos dinámicas*, que crecen y se reducen en tiempo de ejecución. Las *listas enlazadas* son colecciones de elementos de datos "alineados en una fila"; pueden insertarse y eliminarse elementos en cualquier parte de una lista enlazada. Las *pilas* son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos sólo en un extremo de una pila: su *parte superior*. Las *colas* representan líneas de espera; se insertan elementos en la parte final (conocida como *rabo*) de una cola y se eliminan elementos de su parte inicial (conocida como *cabeza*). Los *árboles binarios* facilitan la búsqueda y ordenamiento de los datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos, la compilación de expresiones en lenguaje máquina y muchas otras aplicaciones interesantes.

Hablaremos sobre cada uno de estos tipos de estructuras de datos e implementaremos programas para crearlas y manipularlas. Utilizaremos clases, herencia y composición para crear y empaquetar estas estructuras de datos, para reutilizarlas y darles mantenimiento. En el capítulo 21 y en el capítulo 22 hablaremos sobre las clases predefinidas de Java para implementar las estructuras de datos que describiremos en este capítulo.

Los ejemplos que se presentan aquí son programas prácticos que pueden utilizarse en cursos más avanzados y en aplicaciones industriales. Los ejercicios incluyen una vasta colección de aplicaciones útiles.

Le recomendamos que trate de realizar el proyecto principal descrito en la sección especial titulada *Construya su propio compilador*. Ha estado utilizando un compilador de Java para traducir sus programas de Java en códigos de bytes, para poder ejecutar estos programas en su computadora. En este proyecto creará su propio compilador funcional. Este compilador leerá un archivo de instrucciones escritas en un lenguaje de alto nivel simple pero poderoso, similar a las primeras versiones del popular lenguaje Basic. Su compilador traducirá estas instrucciones en un archivo de instrucciones de Lenguaje máquina Simpletron (LMS); éste es el lenguaje que aprendió en la sección especial del capítulo 7, *Construya su propia computadora*. ¡Su programa Simulador de Simpletron ejecutará entonces el programa LMS producido por su compilador! Al implementar este proyecto mediante el uso de una metodología orientada a objetos, usted recibirá una maravillosa oportunidad para poner en práctica la mayor parte de lo que ha aprendido en este libro. La sección especial lo lleva cuidadosamente a través de las especificaciones del lenguaje de alto nivel, y describe los algoritmos que usted necesitará para convertir cada instrucción, en lenguaje de alto nivel, a instrucciones de lenguaje máquina. Si disfruta de los retos, tal vez pueda tratar de realizar las diversas mejoras tanto al compilador como al Simulador Simpletron, las cuales se sugieren en los ejercicios.

20.2 Clases autorreferenciadas

Una *clase autorreferenciada* contiene una variable de instancia que hace referencia a otro objeto del mismo tipo de clase. Por ejemplo, la declaración:

http://librosysolucionarios.net


```
class Nodo {
   private int datos;
   private Nodo siguienteNodo; // referencia al siguiente nodo enlazado
   public Nodo( int datos )                        { /* cuerpo del constructor */ }
   public void establecerDatos( int datos )        { /* cuerpo del método */ }
   public int obtenerDatos()                       { /* cuerpo del método */ }
   public void establecerSiguiente( Nodo siguiente ) { /* cuerpo del método */ }
   public Nodo obtenerSiguiente()                  { /* cuerpo del método */ }
}
```

declara la clase `Nodo`, la cual tiene dos variables de instancia `private`: la variable entera `datos` y la referencia `Nodo` llamada `siguienteNodo`. El campo `siguienteNodo` hace referencia a un objeto de la clase `Nodo`, un objeto de la misma clase que se está declarando aquí; es por ello que se utiliza el término "clase autorreferenciada". El campo `siguienteNodo` es un *enlace*; "vincula a un objeto de tipo `Nodo` con otro objeto del mismo tipo. El tipo `Nodo` también tiene cinco métodos: un constructor que recibe un entero para inicializar a `datos`, un método `establecerDatos` para establecer el valor de `datos`, un método `obtenerDatos` para devolver el valor de `datos`, un método `establecerSiguiente` para establecer el valor de `siguienteNodo` y un método `obtenerSiguiente` para devolver una referencia al siguiente nodo.

Los programas pueden enlazar objetos autorreferenciados entre sí para formar estructuras de datos útiles como listas, colas, pilas y árboles. En la figura 20.1 se muestran dos objetos autorreferenciados, enlazados entre sí para formar una lista. Una barra diagonal inversa (que representa una referencia `null`) se coloca en el miembro de enlace del segundo objeto autorreferenciado para indicar que el enlace no hace referencia a otro objeto.[1] Por lo general, una referencia `null` indica el final de una estructura de datos.[2]

20.3 Asignación dinámica de memoria

Para crear y mantener estructuras dinámicas de datos se requiere la *asignación dinámica de memoria*; la habilidad para que un programa obtenga más espacio de memoria en tiempo de ejecución, para almacenar nuevos nodos y para liberar el espacio que ya no se necesita. Recuerde que los programas de Java no liberan explícitamente la memoria asignada en forma dinámica. En vez de ello, Java realiza la recolección automática de basura en los objetos que ya no son referenciados en un programa.

El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de espacio en disco disponible en un sistema con memoria virtual. A menudo, los límites son mucho más pequeños ya que la memoria disponible de la computadora debe compartirse entre muchas aplicaciones.

La expresión de declaración y creación de instancia de clase:

```
Nodo nodoParaAgregar = new Nodo( 10 ); // 10 son los datos de nodoParaAgregar
```

asigna la memoria para almacenar un objeto `Nodo` y devuelve una referencia al objeto, que se asigna a `nodoParaAgregar`. Si no hay disponible suficiente memoria, la expresión lanza una excepción `OutOfMemoryError`.

Las siguientes secciones hablan sobre listas, pilas, colas y árboles que utilizan asignación dinámica de memoria y clases autorreferenciadas para crear estructuras de datos dinámicas.

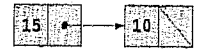


Figura 20.1 Objetos de una clase autorreferenciada enlazados entre sí.

1. La barra diagonal inversa es para fines ilustrativos; no corresponde al carácter de barra diagonal en Java.
2. Hay otras formas de representar el final de una estructura de datos, que se encuentran más allá del alcance de este libro.

20.4 Listas enlazadas

Una *lista enlazada* es una colección lineal (es decir, una secuencia) de objetos de una clase autorreferenciada, conocidos como *nodos*, que están conectados por *enlaces* de referencia; es por ello que se utiliza el término lista "enlazada". Por lo general, un programa accede a una lista enlazada mediante una referencia al primer nodo en la lista. El programa accede a cada nodo subsiguiente a través de la referencia de enlace almacenada en el nodo anterior. Por convención, la referencia de enlace en el último nodo de una lista se establece en `null` para indicar el final de la lista. Los datos se almacenan en forma dinámica en una lista enlazada; el programa crea cada nodo según sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo referencias a objetos de otras clases. Las pilas y las colas son también estructuras de datos lineales y, como veremos, son versiones restringidas de las listas enlazadas. Los árboles son estructuras de datos no lineales.

Pueden almacenarse listas de datos en los arreglos, pero las listas enlazadas ofrecen varias ventajas. Una lista enlazada es apropiada cuando el número de elementos de datos que se van a representar en la estructura de datos es impredecible. Las listas enlazadas son dinámicas, por lo que la longitud de una lista puede incrementarse o reducirse, según sea necesario. Sin embargo, el tamaño de un arreglo "convencional" en Java no puede alterarse, ya que el tamaño del arreglo se fija en el momento en que el programa lo crea. Los arreglos "convencionales" pueden llenarse. Las listas enlazadas se llenan sólo cuando el sistema no tiene suficiente memoria para satisfacer las peticiones de asignación dinámica de almacenamiento. El paquete `java.util` contiene la clase `LinkedList` para implementar y manipular listas enlazadas que crezcan y se reduzcan durante la ejecución del programa. Hablaremos sobre la clase `LinkedList` en el capítulo 22.

Tip de rendimiento 20.1



Un arreglo puede declararse de manera que contenga más elementos que el número de elementos esperados, pero esto desperdicia memoria. Las listas enlazadas proporcionan una mejor utilización de memoria en estas situaciones. Las listas enlazadas permiten al programa adaptarse a las necesidades de almacenamiento en tiempo de ejecución.

Tip de rendimiento 20.2



La inserción en una lista enlazada es rápida; sólo hay que modificar dos referencias (después de localizar el punto de inserción). Todos los objetos nodo existentes permanecen en sus posiciones actuales en memoria.

Las listas enlazadas pueden mantenerse en orden con sólo insertar cada nuevo elemento en el punto apropiado de la lista. (Claro que lleva tiempo localizar el punto de inserción apropiado.) Los elementos existentes en la lista no necesitan moverse.

Tip de rendimiento 20.3



La inserción y la eliminación en un arreglo ordenado puede llevar mucho tiempo; todos los elementos que van después del elemento insertado o eliminado deben desplazarse apropiadamente.

Por lo general, los nodos de las listas enlazadas no se almacenan contiguamente en memoria, sino que son adyacentes en forma lógica. La figura 20.2 muestra una lista enlazada con varios nodos. Este diagrama presenta una *lista de enlace simple* (cada nodo contiene una referencia al siguiente nodo en la lista). A menudo, las listas enlazadas se implementan como listas doblemente enlazadas (cada nodo contiene una referencia al siguiente nodo en la lista y una referencia al nodo anterior en la lista). La clase `LinkedList` de Java es una implementación de lista doblemente enlazada.

Tip de rendimiento 20.4



Por lo general, los elementos de un arreglo están contiguos en memoria. Esto permite un acceso inmediato a cualquier elemento del arreglo, ya que la dirección de cualquier elemento puede calcularse directamente como su desplazamiento a partir del inicio del arreglo. Las listas enlazadas no permiten dicho acceso inmediato a sus elementos; para acceder a ellos se tiene que recorrer la lista desde su parte inicial (o desde la parte final en una lista doblemente enlazada).

El programa de la figuras 20.3 a 20.5 utiliza un objeto de nuestra clase `Lista` para manipular una lista de objetos misceláneos. El programa consta de cuatro clases: `NodoLista` (figura 20.3, líneas 6 a 37), `Lista` (figura 20.3, líneas 40 a 149), `ExcepcionListaVacia` (figura 20.4) y `PruebaLista` (figura 20.5). Las clases `Lista`, `NodoLista` y `ExcepcionListaVacia` están colocadas en el paquete `com.deitel.`

http://librosysolucionarios.net

`cpej5.cap20`[3], para que puedan reutilizarse a lo largo de este capítulo. Hay una lista enlazada de objetos `NodoLista` encapsulada en cada objeto `Lista`.

La clase `NodoLista` (figura 20.3, líneas 6 a 37) declara los campos de acceso del paquete llamados `datos` y `siguienteNodo`. El campo `datos` es una referencia `Object`, por lo que puede hacer referencia a cualquier objeto. El miembro `siguienteNodo` de `NodoLista` almacena una referencia al siguiente objeto `NodoLista` en la lista enlazada (o `null`, si el nodo es el último en la lista).

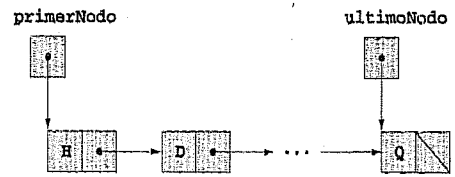


Figura 20.2 Representación gráfica de una lista enlazada.

```
1  // Fig. 20.3: Lista.java
2  // Definiciones de las clases NodoLista y Lista.
3  package com.deitel.cpej5.cap20;
4
5  // clase para representar un nodo en una lista
6  class NodoLista {
7
8     // miembros de acceso del paquete; Lista puede utilizarlos directamente
9     Object datos;
10    NodoLista siguienteNodo;
11
12    // crear un objeto NodoLista que haga referencia a objeto
13    NodoLista( Object objeto )
14    {
15       this( objeto, null );
16    }
17
18    // crear objeto NodoLista que haga referencia al objeto Object y al siguiente
         objeto NodoLista
19    NodoLista( Object objeto, NodoLista nodo )
20    {
21       datos = objeto;
22       siguienteNodo = nodo;
23    }
24
25    // devolver referencia a datos en nodo
26    Object obtenerObjeto()
27    {
28       return datos; // devolver objeto Object en este nodo
29    }
30
```

Figura 20.3 Declaraciones de las clases `NodoLista` y `Lista`. (Parte 1 de 4.)

3. Muchas de las clases en este capítulo se declaran en el paquete `com.deitel.cpej5.cap20`. Cada una de esas clases debe compilarse con la opción de línea de comandos `-d` en `javac`. Al compilar las clases que no se encuentran en este paquete y al ejecutar los programas, asegúrese de utilizar la opción `-classpath` con `javac` y `java`, respectivamente.

```
31     // devolver referencia al siguiente nodo en la lista
32     NodoLista obtenerSiguiente()
33     {
34        return siguienteNodo; // obtener el siguiente nodo
35     }
36
37  } // fin de la clase NodoLista
38
39  // definición de la clase Lista
40  public class Lista {
41     private NodoLista primerNodo;
42     private NodoLista ultimoNodo;
43     private String nombre;  // cadena como "lista", utilizada para imprimir
44
45     // construir Lista vacía con "lista" como nombre
46     public Lista()
47     {
48        this( "lista" );
49     }
50
51     // construir una Lista vacía con un nombre
52     public Lista( String nombreLista )
53     {
54        nombre = nombreLista;
55        primerNodo = ultimoNodo = null;
56     }
57
58     // insertar objeto Object al frente de Lista
59     public synchronized void insertarAlFrente( Object elementoInsertar )
60     {
61        if ( estaVacia() ) // primerNodo y ultimoNodo hacen referencia al mismo
          objeto Object
62           primerNodo = ultimoNodo = new NodoLista( elementoInsertar );
63
64        else // primerNodo hace referencia a un nuevo nodo
65           primerNodo = new NodoLista( elementoInsertar, primerNodo );
66     }
67
68     // insertar objeto Object al final de la Lista
69     public synchronized void insertarAlFinal( Object elementoInsertar )
70     {
71        if ( estaVacia() ) // primerNodo y ultimoNodo hacen referencia al mismo objeto Object
72           primerNodo = ultimoNodo = new NodoLista( elementoInsertar );
73
74        else // el objeto siguienteNodo que va después de ultimoNodo hace referencia
          a un nuevo nodo
75           ultimoNodo = ultimoNodo.siguienteNodo = new NodoLista( elementoInsertar );
76     }
77
78     // eliminar primer nodo de la Lista
79     public synchronized Object eliminarDelFrente() throws ExcepcionListaVacia
80     {
81        if ( estaVacia() ) // lanzar excepción si Lista está vacía
82           throw new ExcepcionListaVacia( nombre );
83
84        Object elementoEliminado = primerNodo.datos; // recuperar los datos que se van a eliminar
85
```

http://librosysolucionarios.net

Figura 20.3 Declaraciones de las clases **NodoLista** y **Lista**. (Parte 2 de 4.)


```
86        // actualizar las referencias primerNodo y ultimoNodo
87        if ( primerNodo == ultimoNodo )
88           primerNodo = ultimoNodo = null;
89        else
90           primerNodo = primerNodo.siguienteNodo;
91
92        return elementoEliminado; // devolver datos del nodo eliminado
93
94     } // fin del método eliminarDelFrente
95
96     // eliminar último nodo de la Lista
97     public synchronized Object eliminarDelFinal() throws ExcepcionListaVacia
98     {
99        if ( estaVacia() ) // lanzar excepción si Lista está vacía
100          throw new ExcepcionListaVacia( nombre );
101
102       Object elementoEliminado = ultimoNodo.datos; // recuperar los datos que se
          van a eliminar
103
104       // actualizar las referencias primerNodo y ultimoNodo
105       if ( primerNodo == ultimoNodo )
106          primerNodo = ultimoNodo = null;
107
108       else { // localizar nuevo último nodo
109          NodoLista actual = primerNodo;
110
111          // iterar mientras nodo actual no haga referencia al ultimoNodo
112          while ( actual.siguienteNodo != ultimoNodo )
113             actual = actual.siguienteNodo;
114
115          ultimoNodo = actual; // actual es el nuevo ultimoNodo
116          actual.siguienteNodo = null;
117       }
118
119       return elementoEliminado; // devolver datos del nodo eliminado
120
121    } // fin del método eliminarDelFinal
122
123    // determinar si la lista está vacía
124    public synchronized boolean estaVacia()
125    {
126       return primerNodo == null; // devolver true si la Lista está vacía
127    }
128
129    // mostrar el contenido de la Lista
130    public synchronized void imprimir()
131    {
132       if ( estaVacia() ) {
133          System.out.println( nombre + " vacia" );
134          return;
135       }
136
137       System.out.print( "La " + nombre + " es: " );
138       NodoLista actual = primerNodo;
139
140       // mientras no se encuentre al final de la lista, imprimir los datos del nodo actual
```

Figura 20.3 Declaraciones de las clases **NodoLista** y **Lista**. (Parte 3 de 4.)

```
141         while ( actual != null ) {
142            System.out.print( actual.datos.toString() + " " );
143            actual = actual.siguienteNodo;
144         }
145
146         System.out.println( "\n" );
147      }
148
149   } // fin de la clase Lista
```

Figura 20.3 Declaraciones de las clases **NodoLista** y **Lista**. (Parte 4 de 4.)

En las líneas 41 y 42 de la clase **Lista** (líneas 40 a 149) se declaran referencias al primer y último objetos **NodoLista** en un objeto **Lista** (**primerNodo** y **ultimoNodo**, respectivamente). Los constructores (líneas 46 a 49 y 52 a 56) inicializan ambas referencias con **null**. Los métodos más importantes de la clase **Lista** son los métodos **synchronized** **insertarAlFrente** (líneas 59 a 66), **insertarAlFinal** (líneas 69 a 76), **eliminarDelFrente** (líneas 79 a 94) y **eliminarDelFinal** (líneas 97 a 121). Estos métodos se declaran como **synchronized**, para que los objetos **Lista** puedan usarse con seguridad en los subprocesos de un programa con subprocesamiento múltiple. Si un subproceso está modificando el contenido de un objeto **Lista**, ningún otro subproceso podrá modificar el mismo objeto **Lista** al mismo tiempo. El método **estaVacia** (líneas 124 a 127) es un *método predicado*, el cual determina si la lista está vacía (es decir, si la referencia al primer nodo de la lista es **null**). Los métodos predicado generalmente evalúan una condición y no modifican el objeto en el que son llamados. Si la lista está vacía, el método **estaVacia** devuelve **true**; en caso contrario, devuelve **false**. El método **imprimir** (líneas 130 a 147) muestra el contenido de la lista. Tanto **estaVacia** como **imprimir** son también métodos **synchronized**, para asegurar que el estado de la lista no cambie mientras estos métodos se ejecuten, lo cual ayuda a prevenir resultados incorrectos. Después de la figura 20.5 hay una discusión detallada sobre los métodos de **Lista**.

El método **main** de la clase **PruebaLista** (figura 20.5) crea una lista de objetos, inserta objetos al principio de la lista utilizando el método **insertarAlFrente**, inserta objetos al final de la lista utilizando el método **insertarAlFinal**, elimina objetos de la parte frontal de la lista utilizando el método **eliminarDelFrente** y elimina objetos de la parte final de la lista utilizando el método **eliminarDelFinal**. Después de cada operación de inserción y eliminación, **PruebaLista** llama al método **Lista** llamado **imprimir** para mostrar el contenido actual de la lista. Si se trata de eliminar un elemento de una lista vacía se lanza una excepción del tipo **ExcepcionListaVacia** (figura 20.4), de manera que las llamadas a los métodos **eliminarDelFrente** y **eliminarDelFinal** se colocan en un bloque **try** que va seguido de un manejador de excepciones apropiado.

```
1  // Fig. 20.4: ExcepcionListaVacia.java
2  // Definición de la clase ExcepcionListaVacia.
3  package com.deitel.cpej5.cap20;
4
5  public class ExcepcionListaVacia extends RuntimeException {
6
7     // constructor sin argumentos
8     public ExcepcionListaVacia()
9     {
10       this( "Lista" );  // llamar al otro constructor de ExcepcionListaVacia
11    }
12
13    // constructor
14    public ExcepcionListaVacia( String nombre )
```

Figura 20.4 Declaración de la clase **ExcepcionListaVacia**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
15    {
16       super( nombre + " esta vacia" );  // llamar al constructor de la superclase
17    }
18
19 } // fin de la clase ExcepcionListaVacia
```

Figura 20.4 Declaración de la clase **ExcepcionListaVacia**. (Parte 2 de 2.)

```
1  // Fig. 20.5: PruebaLista.java
2  // Clase PruebaLista para demostrar las herramientas de la clase Lista.
3  import com.deitel.cpej5.cap20.Lista;
4  import com.deitel.cpej5.cap20.ExcepcionListaVacia;
5
6  public class PruebaLista {
7
8     public static void main( String args[] )
9     {
10       Lista lista = new Lista();  // crear el contenedor Lista
11
12       // objetos a guardar en la lista
13       Boolean bool = Boolean.TRUE;
14       Character caracter = new Character( '$' );
15       Integer entero = new Integer( 34567 );
16       String cadena = "hola";
17
18       // insertar referencias a objetos en la lista
19       lista.insertarAlFrente( bool );
20       lista.imprimir();
21       lista.insertarAlFrente( caracter );
22       lista.imprimir();
23       lista.insertarAlFinal( entero );
24       lista.imprimir();
25       lista.insertarAlFinal( cadena );
26       lista.imprimir();
27
28       // eliminar objetos de la lista; imprimir después de cada remoción
29       try {
30          Object objetoEliminado = lista.eliminarDelFrente();
31          System.out.println( objetoEliminado.toString() + " eliminado" );
32          lista.imprimir();
33
34          objetoEliminado = lista.eliminarDelFrente();
35          System.out.println( objetoEliminado.toString() + " eliminado" );
36          lista.imprimir();
37
38          objetoEliminado = lista.eliminarDelFinal();
39          System.out.println( objetoEliminado.toString() + " eliminado" );
40          lista.imprimir();
41
42          objetoEliminado = lista.eliminarDelFinal();
43          System.out.println( objetoEliminado.toString() + " eliminado" );
44          lista.imprimir();
45
46       } // fin del bloque try
```

Figura 20.5 Manipulaciones de listas enlazadas. (Parte 1 de 2.)

```
47
48          // atrapar excepción si se intenta eliminar en una Lista vacía
49          catch ( ExcepcionListaVacia excepcionListaVacia ) {
50             excepcionListaVacia.printStackTrace();
51          }
52       }
53
54    } // fin de la clase PruebaLista
```

```
La lista es: true

La lista es: $ true

La lista es: $ true 34567

La lista es: $ true 34567 hola

$ eliminado
La lista es: true 34567 hola

true eliminado
La lista es: 34567 hola

hola eliminado
La lista es: 34567

34567 eliminado
Lista vacia
```

Figura 20.5 Manipulaciones de listas enlazadas. (Parte 2 de 2.)

Ahora hablaremos detalladamente sobre cada uno de los métodos de la clase `Lista` (figura 20.3) y proporcionaremos diagramas que muestren las manipulaciones de referencia realizadas por los métodos `insertarAlFrente`, `insertarAlFinal`, `eliminarDelFrente` y `eliminarDelFinal`. El método `insertarAlFrente` (líneas 59 a 66 de la figura 20.3) coloca un nuevo nodo al frente de la lista. Los pasos son:

1. Llamar a `estaVacia` para determinar si la lista está vacía (línea 61).
2. Si la lista está vacía, asignar `primerNodo` y `ultimoNodo` al nuevo `NodoLista` que se inicializó con `elementoInsertar` (línea 62). El constructor de `NodoLista` en las líneas 13 a 16 llama al constructor de `NodoLista` en las líneas 19 a 23 para establecer la variable de instancia `datos`, para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y para establecer la referencia `siguienteNodo` en `null`, ya que éste es el primer y último nodo en la lista.
3. Si la lista no está vacía, el nuevo nodo se "enlaza" en la lista asignando a `primerNodo` un nuevo objeto `NodoLista`, e inicializando ese objeto con `elementoInsertar` y `primerNodo` (línea 65). Cuando se ejecuta el constructor de `NodoLista` (líneas 19 a 23), establece la variable de instancia `datos` para que haga referencia al `elementoInsertar` que se pasa como argumento, y realiza la inserción asignando a la referencia `siguienteNodo` del nuevo nodo el objeto `NodoLista` que se pasa como argumento y que anteriormente era el primer nodo.

En la figura 20.6, la parte (a) muestra una lista y un nuevo nodo durante la operación `insertarAlFrente` y antes de que el programa enlace el nuevo nodo a la lista. Las flechas punteadas en la parte (b) ilustran el paso 3 de la operación `insertarAlFrente`, en donde se permite al nodo que contiene `12` convertirse en el primer nuevo nodo en la lista.

El método `insertarAlFinal` (líneas 69 a 76 de la figura 20.3) coloca un nuevo nodo al final de la lista. Los pasos son:

1. Llamar a `estaVacia` para determinar si la lista está vacía (línea 71).

http://librosysolucionarios.net

2. Si la lista está vacía, asignar `primerNodo` y `ultimoNodo` al nuevo `NodoLista` que se inicializó con `elementoInsertar` (línea 72). El constructor de `NodoLista` en las líneas 13 a 16 llama al constructor de las líneas 19 a 23 para establecer la variable de instancia `datos`, para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y para establecer la referencia `siguienteNodo` en `null`.
3. Si la lista no está vacía, en la línea 75 se enlaza el nuevo nodo a la lista, asignando a `ultimoNodo` y a `ultimoNodo.siguienteNodo` la referencia al nuevo `NodoLista` que se inicializó con `elementoInsertar`. El constructor de `NodoLista` (líneas 13 a 16) establece la variable de instancia `datos` para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y establece la referencia `siguienteNodo` en `null`, ya que éste es el último nodo en la lista.

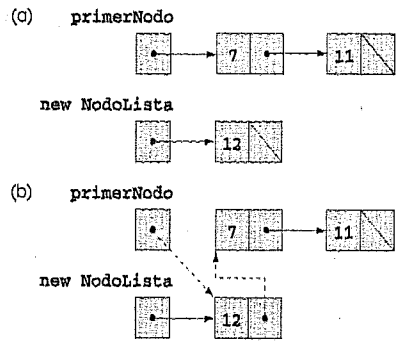


Figura 20.6 Representación gráfica de la operación `insertarAlFrente`.

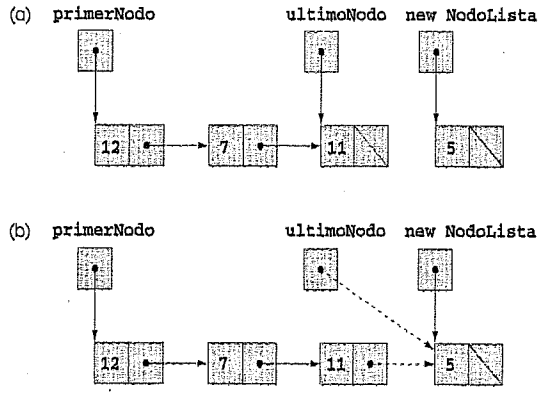


Figura 20.7 Representación gráfica de la operación `insertarAlFinal`.

En la figura 20.7, la parte (a) muestra una lista y un nuevo nodo durante la operación `insertarAlFrente` y antes de que el programa enlace el nuevo nodo a la lista. Las flechas punteadas en la parte (b) ilustran el paso 3 del método `insertarAlFinal`, el cual agrega el nuevo nodo al final de una lista que no está vacía.

El método `eliminarDelFrente` (líneas 79 a 94 de la figura 20.3) elimina el primer nodo de la lista y devuelve una referencia a los datos eliminados. El método lanza una excepción `ExcepcionListaVacia`

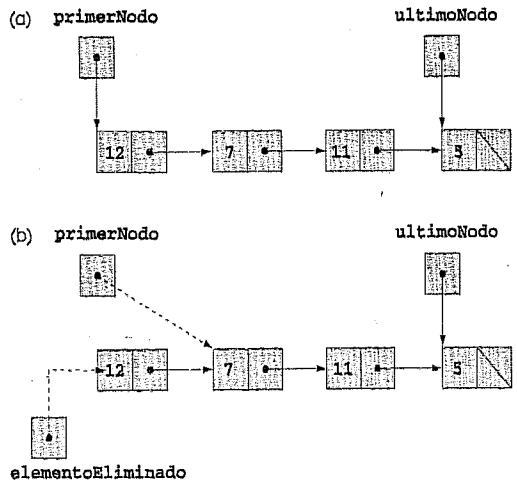


Figura 20.8 Representación gráfica de la operación `eliminarDelFrente`.

(líneas 81 y 82) si la lista está vacía cuando el programa llama a este método. De no ser así, el método devuelve una referencia a los datos eliminados. Los pasos son:

1. Asignar `primerNodo.datos` (los datos que se van a eliminar de la lista) a la referencia `elementoEliminado` (línea 84).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 87), quiere decir que la lista sólo tiene un elemento en ese momento. Por lo tanto, el método establece a `primerNodo` y `ultimoNodo` en `null` (línea 88) para eliminar el nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, entonces el método deja la referencia `ultimoNodo` como está y asigna el valor de `primerNodo.siguienteNodo` a `primerNodo` (línea 90). Por lo tanto, `primerNodo` hace referencia al nodo que era anteriormente el segundo nodo en la lista.
4. Devolver la referencia `elementoEliminado` (línea 92).

En la figura 20.8, la parte (a) ilustra la lista antes de la operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de referencias.

El método `eliminarDelFinal` (líneas 97 a 121 de la figura 20.3) elimina el último nodo de una lista y devuelve una referencia a los datos eliminados. El método lanza una excepción `ExcepcionListaVacia` (líneas 99 y 100) si la lista está vacía cuando el programa llama a este método. Los pasos son:

1. Asignar `ultimoNodo.datos` (los datos que se van a eliminar de la lista) a `elementoEliminado` (línea 102).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 105), quiere decir que la lista sólo tiene un elemento en ese momento. Por lo tanto, en la línea 106 se establece a `primerNodo` y `ultimoNodo` en `null` para eliminar ese nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, crear la referencia `NodoLista` llamada `actual` y asignarla a `primerNodo` (línea 109).
4. Ahora hay que "recorrer la lista" con `actual` hasta que haga referencia al nodo que esté antes del último nodo. El ciclo `while` (líneas 112 y 113) asigna `actual.siguienteNodo` a `actual`, siempre y cuando `actual.siguienteNodo` (el siguiente nodo en la lista) no sea `ultimoNodo`.
5. Después de localizar el penúltimo nodo, asignar `actual` a `ultimoNodo` (línea [illegible]) para actualizar cuál nodo es el último en la lista.

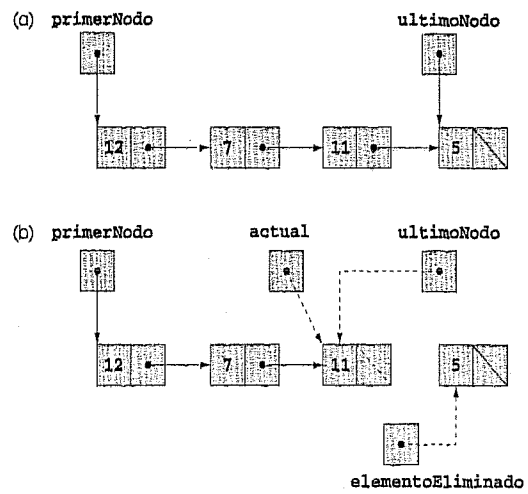


Figura 20.9 Representación gráfica de la operación `eliminarDelFinal`.

6. Establecer `actual.siguienteNodo` en `null` (línea 116) para eliminar el último nodo de la lista y terminarla con el nodo actual.
7. Devolver la referencia `elementoEliminado` (línea 119).

En la figura 20.9, la parte (a) ilustra la lista antes de la operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de referencias.

El método `imprimir` (líneas 130 a 147) determina primero si la lista está vacía (líneas 132 a 135). De ser así, `imprimir` muestra un mensaje indicando que la lista está vacía y devuelve el control al método que hizo la llamada. En caso contrario, `imprimir` muestra en pantalla los datos en la lista. En la línea 138 se crea la referencia `NodoLista` llamada `actual` y se inicializa con `primerNodo`. Mientras que `actual` no sea `null`, hay más elementos en la lista. Por lo tanto, en la línea 142 se muestra en pantalla una representación de cadena de `actual.datos`. En la línea 143 se avanza al siguiente nodo en la lista mediante la asignación del valor de la referencia `actual.siguienteNodo` a `actual`. Este algoritmo de impresión es idéntico para listas enlazadas, pilas y colas.

20.5 Pilas

Una *pila* es una versión restringida de una lista enlazada;[4] pueden agregarse y eliminarse nuevos nodos en una pila solamente desde su parte superior. Por esta razón, a una pila se le conoce como estructura de datos *UEPS (último en entrar, primero en salir)*. El miembro de enlace en el nodo inferior (es decir, el último) de la pila se establece en `null` para indicar el fondo de la pila.

Los métodos básicos para manipular una pila son ***push*** (empujar) y ***pop*** (sacar). El método `push` agrega un nuevo nodo a la parte superior de la pila. El método `pop` elimina un nodo de la parte superior de la pila y devuelve los datos del nodo que se quitó.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando un programa llama a un método, el método llamado debe saber cómo regresar su invocador, por lo que la dirección de retorno del método que hizo la llamada se mete en la *pila de ejecución del programa*. Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila en el orden "último en entrar, primero en salir", para que

http://librosysolucionarios.net

4. Una pila no tiene que implementarse utilizando una lista enlazada.

cada método pueda regresar a su invocador. Las pilas soportan llamadas recursivas a métodos de la misma manera que para las llamadas no recursivas convencionales.

La pila de ejecución del programa también contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Cuando el método regresa a su invocador, la memoria para las variables locales de ese método se saca de la pila y esas variables dejan de ser conocidas por el programa.[5]

Los compiladores utilizan pilas para evaluar expresiones aritméticas y generar código en lenguaje máquina para procesar las expresiones. Los ejercicios en este capítulo exploran varias aplicaciones de las pilas, incluyendo el utilizarlas para desarrollar un compilador funcional completo. Además, el paquete `java.util` contiene la clase `Stack` (vea el capítulo 21) para implementar y manipular pilas que pueden crecer y reducirse en tamaño durante la ejecución del programa.

Tomemos ventaja de la estrecha relación entre las listas y las pilas para implementar una clase de pila, mediante la reutilización de una clase de lista. Mostraremos dos formas distintas de reutilización: primero vamos a implementar la clase de pila extendiendo a la clase `Lista` de la figura 20.3. Después implementaremos una clase de pila con la misma funcionalidad por medio de la composición, incluyendo una referencia a un objeto `Lista` como una variable de instancia privada de una clase de pila. Las estructuras de datos lista, pila y cola en este capítulo se implementan para almacenar referencias `Object`, para exhortar a que se reutilicen en el futuro. Así, puede guardarse cualquier tipo de objeto en una lista, pila o cola.

Clase de pila que hereda de `Lista`

En la aplicación de las figuras 20.10 y 20.11 se crea una clase de pila extendiendo a la clase `Lista` de la figura 20.3. Queremos que la pila tenga los métodos `push`, `pop`, `estaVacia` e `imprimir`. En esencia, éstos son los métodos `insertarAlFrente`, `eliminarDelFrente`, `estaVacia` e `imprimir` de la clase `Lista`. Desde luego que la clase `Lista` contiene otros métodos (como `insertarAlFinal` y `eliminarDelFinal`) que preferiríamos no estuvieran accesibles mediante la interfaz `public` para la clase de pila. Es importante recordar que todos los métodos en la interfaz `public` de la clase `Lista` también son métodos `public` de la subclase `HerenciaPila` (figura 20.10). Para implementar los métodos de la pila, haremos que cada método de `HerenciaPila` llame al método apropiado de `Lista`; el método `push` llama a `insertarAlFrente` y el método `pop` llama a `eliminarDelFrente`. Los clientes de la clase `HerenciaPila` pueden llamar a los métodos `estaVacia` e `imprimir`, ya que son heredados de `Lista`. La clase `HerenciaPila` se declara como parte del paquete `com.deitel.cpej5.cap20` para fines de reutilización. Observe que `HerenciaPila` no importa a `Lista`, ya que ambas clases se encuentran en el mismo paquete.

```
1  // Fig. 20.10: HerenciaPila.java
2  // Derivada de la clase Lista.
3  package com.deitel.cpej5.cap20;
4
5  public class HerenciaPila extends Lista {
6
7     // construir pila
8     public HerenciaPila()
9     {
10       super( "pila" );
11    }
12
13    // agregar objeto a la pila
14    public synchronized void push( Object objeto )
15    {
16       insertarAlFrente( objeto );
17    }
18
19    // eliminar objeto de la pila
20    public synchronized Object pop() throws ExcepcionListaVacia
21    {
22       return eliminarDelFrente();
23    }
24
25 } // fin de la clase HerenciaPila
```

Figura 20.10 `HerenciaPila` extiende a la clase `Lista`. (Parte 1 de 2.)

5. Si la variable local es una referencia, la cuenta de referencias para el objeto al que se está haciendo referencia se incrementa en 1. Si la cuenta de referencias llega a cero, el objeto puede marcarse para la recolección de basura.

http://librosysolucionarios.net

El método `main` de la clase `PruebaHerenciaPila` (figura 20.11) crea un objeto de la clase `HerenciaPila` llamado `pila` (línea 10). El programa mete en la pila (líneas 19, 21, 23 y 25) un objeto `Boolean` que contiene `true`, un objeto `Character` que contiene `$`, un objeto `Integer` que contiene `34567` y un objeto `String` que contiene `hola`. En las líneas 32 a 36 se sacan objetos de la pila en un ciclo `while` infinito. Si el método `pop` se invoca en una pila vacía, el método lanza una excepción `ExcepcionListaVacia`. En este caso, el programa muestra el rastreo de la pila de la excepción, que muestran los métodos en la pila de ejecución del programa al momento en que ocurrió la excepción. Observe que el programa utiliza el método `imprimir` (heredado de `Lista`) para mostrar el contenido de la pila.

```
1  // Fig. 20.11: PruebaHerenciaPila.java
2  // La clase PruebaHerenciaPila.
3  import com.deitel.cpej5.cap20.HerenciaPila;
4  import com.deitel.cpej5.cap20.ExcepcionListaVacia;
5
6  public class PruebaHerenciaPila {
7
8     public static void main( String args[] )
9     {
10       HerenciaPila pila = new HerenciaPila();
11
12       // crear objetos que se van a almacenar en la pila
13       Boolean bool = Boolean.TRUE;
14       Character caracter = new Character( '$' );
15       Integer entero = new Integer( 34567 );
16       String cadena = "hola";
17
18       // usar método push
19       pila.push( bool );
20       pila.imprimir();
21       pila.push( caracter );
22       pila.imprimir();
23       pila.push( entero );
24       pila.imprimir();
25       pila.push( cadena );
26       pila.imprimir();
27
28       // eliminar elementos de la pila
29       try {
30          Object objetoEliminado = null;
31
32          while ( true ) {
```

Figura 20.11 Programa para manipular una pila. (Parte 1 de 2.)

```
33                objetoEliminado = pila.pop(); // usar método pop
34                System.out.println( objetoEliminado.toString() + " sacado" );
35                pila.imprimir();
36             }
37          }
38
39          // atrapar excepción si pila queda vacía al sacar elemento
40          catch ( ExcepcionListaVacia excepcionListaVacia ) {
41             excepcionListaVacia.printStackTrace();
42          }
43       }
44
45    } // fin de la clase PruebaHerenciaPila
```

```
La pila es: true

La pila es: 5 true

La pila es: 34567 5 true

La pila es: hola 34567 5 true

hola sacado
La pila es: 34567 5 true

34567 sacado
La pila es: 5 true

5 sacado
La pila es: true

true sacado
Pila vacia

com.deitel.cpej5.cap20.ExcepcionListaVacia: pila esta vacia
        at com.deitel.cpej5.cap20.Lista.eliminarDelFrente(Lista.java:82)
        at com.deitel.cpej5.cap20.HerenciaPila.pop(HerenciaPila.java:22)
        at PruebaHerenciaPila.main(PruebaHerenciaPila.java:33)
```

Figura 20.11 Programa para manipular una pila. (Parte 2 de 2.)

Clase de pila que contiene una referencia a una `Lista`

Otra manera de implementar una clase de pila es reutilizando una clase de lista mediante la composición. En la figura 20.12 se utiliza una referencia privada a un objeto de la clase **`Lista`** (línea 6) en la declaración de la clase **`ComposicionPila`**. La composición nos permite ocultar los métodos de la clase **`Lista`** que no deben estar en la interfaz pública de nuestra pila, proporcionando métodos de interfaz públicos solamente a los métodos requeridos de **`Lista`**. Esta técnica de implementar cada método de la pila como una llamada a un método de **`Lista`** se conoce como *delegación*; el método invocado de la pila *delega* la llamada al método apropiado de **`Lista`**. Específicamente, **`ComposicionPila`** delega las llamadas a los métodos de **`Lista`** **`insertarAlFrente`**, **`eliminarDelFrente`**, **`estaVacia`** e **`imprimir`**. En este ejemplo no mostramos la clase **`PruebaComposicionPila`**, ya que la única diferencia en este ejemplo es que cambiamos el tipo de la pila, de **`HerenciaPila`** a **`ComposicionPila`** (líneas 3 y 10 de la figura 20.11). La salida es idéntica utilizando cualquier versión de la pila.

http://librosysolucionarios.net


```
1  // Fig. 20.12: ComposicionPila.java
2  // Definición de la clase ComposicionPila con objeto Lista compuesto.
3  package com.deitel.cpej5.cap20;
4
5  public class ComposicionPila {
6     private Lista listaPila;
7
8     // construir pila
9     public ComposicionPila()
10    {
11       listaPila = new Lista( "pila" );
12    }
13
14    // agregar objeto a la pila
15    public synchronized void push( Object objeto )
16    {
17       listaPila.insertarAlFrente( objeto );
18    }
19
20    // eliminar objeto de la pila
21    public synchronized Object pop() throws ExcepcionListaVacia
22    {
23       return listaPila.eliminarDelFrente();
24    }
25
26    // determinar si la pila está vacía
27    public synchronized boolean estaVacia()
28    {
29       return listaPila.estaVacia();
30    }
31
32    // mostrar el contenido de la pila
33    public synchronized void imprimir()
34    {
35       listaPila.imprimir();
36    }
37
38 } // fin de la clase ComposicionPila
```

Figura 20.12 La clase **`ComposicionPila`** utiliza un objeto **`Lista`** compuesto.

20.6 Colas

Otra estructura de datos que se utiliza comúnmente es la *cola*. Una cola es similar a la fila para pagar en un supermercado: el cajero atiende primero a la persona que se encuentra hasta adelante. Los demás clientes entran a la fila sólo por su parte final y esperan a que se les atienda. Los nodos de una cola se eliminan sólo desde el *principio* de la misma y se insertan sólo al *final* de ésta. Por esta razón, a una cola se le conoce como estructura de datos *PEPS (primero en entrar, primero en salir)*. Las operaciones para insertar y eliminar se conocen como ***`enqueue`*** (agregar a la cola) y ***`dequeue`*** (y retirar de la cola).

Las colas tienen muchas aplicaciones en los sistemas computacionales. La mayoría de las computadoras tienen sólo un procesador, por lo que sólo pueden atender a una aplicación a la vez. Cada aplicación que requiere tiempo del procesador se coloca en una cola. La aplicación al frente de la cola es la siguiente que recibe atención. Cada aplicación avanza gradualmente al frente de la cola, a medida que las aplicaciones al frente reciben atención.

Las colas también se utilizan para dar soporte al uso de la *cola de impresión*. Por ejemplo, una sola impresora puede compartirse entre todos los usuarios de la red. Muchos usuarios pueden enviar trabajos a la impreso-

ra, incluso cuando ésta ya se encuentre ocupada. Estos trabajos de impresión se colocan en una cola hasta que la impresora esté disponible. Un programa conocido como *spooler* administra la cola para asegurarse que, a medida que se complete cada trabajo de impresión, se envíe el siguiente trabajo a la impresora.

En las redes computacionales, los paquetes de información también esperan en colas. Cada vez que un paquete llega a un nodo de la red, debe enrutarse hacia el siguiente nodo en la red a través de la ruta hacia el destino final del paquete. El nodo enrutador envía un paquete a la vez, por lo que los paquetes adicionales se ponen en una cola hasta que el enrutador pueda enviarlos.

Un servidor de archivos en una red computacional se encarga de las peticiones de acceso a los archivos de muchos clientes distribuidos en la red. Los servidores tienen una capacidad limitada para dar servicio a las peticiones de los clientes. Cuando se excede esa capacidad, las peticiones de los clientes esperan en colas.

En la figura 20.13 se crea una clase de cola que contiene un objeto de la clase **Lista** (figura 20.3). La clase **Cola** (figura 20.13) proporciona los métodos **enqueue**, **dequeue**, **estaVacia** e **imprimir**. La clase **Lista** contiene otros métodos (por ejemplo, **insertarAlFrente** y **eliminarDelFrente**) que preferiríamos no estuvieran accesibles mediante la interfaz pública para la clase **Cola**. Mediante el uso de la composición, estos métodos en la interfaz pública de la clase **Lista** no son accesibles para los clientes de la clase **Cola**. Cada método de la clase **Cola** llama a un método apropiado de **Lista**; el método **enqueue** llama al método **Lista insertarAlFinal**, el método **dequeue** llama al método **Lista eliminarDelFrente**, el método **estaVacia** llama al método **Lista estaVacia** y el método **imprimir** llama al método **Lista imprimir**. Para fines de reutilización, la clase **Cola** se declara en el paquete **com.deitel.cpej5.cap20**.

```
1  // Fig. 20.13: Cola.java
2  // La clase Cola.
3  package com.deitel.cpej5.cap20;
4
5  public class Cola {
6     private Lista listaCola;
7
8     // construir cola
9     public Cola()
10    {
11       listaCola = new Lista( "cola" );
12    }
13
14    // agregar objeto a la cola
15    public synchronized void enqueue( Object objeto )
16    {
17       listaCola.insertarAlFinal( objeto );
18    }
19
20    // eliminar objeto de la cola
21    public synchronized Object dequeue() throws ExcepcionListaVacia
22    {
23       return listaCola.eliminarDelFrente();
24    }
25
26    // determinar si la cola está vacía
27    public synchronized boolean estaVacia()
28    {
29       return listaCola.estaVacia();
30    }
31
32    // mostrar el contenido de la cola
```

Figura 20.13 La clase **Cola** utiliza la clase **Lista**. (Parte 1 de 2.)


```
33    public synchronized void imprimir()
34    {
35       listaCola.imprimir();
36    }
37
38 } // fin de la clase Cola
```

Figura 20.13 La clase **Cola** utiliza la clase **Lista**. (Parte 2 de 2.)

El método **main** de la clase **PruebaCola** (figura 20.14) crea un objeto de la clase **Cola** llamado **cola**. En las líneas 19, 21, 23 y 25 se agrega a la cola un objeto **Boolean** que contiene **true**, un objeto **Character** que contiene **$**, un objeto **Integer** que contiene **34567** y un objeto **String** que contiene **hola**. En las líneas 32 a 36 se utiliza un ciclo **while** infinito para sacar de la cola los objetos, en el orden "primero en entrar, primero en salir". Cuando la cola está vacía, el método **dequeue** lanza una excepción **ExcepcionListaVacia** y el programa muestra el rastreo de la pila para esa excepción.

```
1  // Fig. 20.14: PruebaCola.java
2  // La clase PruebaCola.
3  import com.deitel.cpej5.cap20.Cola;
4  import com.deitel.cpej5.cap20.ExcepcionListaVacia;
5
6  public class PruebaCola {
7
8     public static void main( String args[] )
9     {
10       Cola cola = new Cola();
11
12       // crear objetos para almacenar en la cola
13       Boolean bool = Boolean.TRUE;
14       Character caracter = new Character( '$' );
15       Integer entero = new Integer( 34567 );
16       String cadena = "hola";
17
18       // usar método enqueue
19       cola.enqueue( bool );
20       cola.imprimir();
21       cola.enqueue( caracter );
22       cola.imprimir();
23       cola.enqueue( entero );
24       cola.imprimir();
25       cola.enqueue( cadena );
26       cola.imprimir();
27
28       // eliminar objetos de la cola
29       try {
30          Object objetoEliminado = null;
31
32          while ( true ) {
33             objetoEliminado = cola.dequeue(); // usar método dequeue
34             System.out.println( objetoEliminado.toString() + " se quito de la cola" );
35             cola.imprimir();
36          }
37       }
```

Figura 20.14 Programa para procesar una cola. (Parte 1 de 2.)

http://librosysolucionarios.net

```
38
39          // procesar excepción si la cola está vacía al eliminar el elemento
40          catch ( ExcepcionListaVacia excepcionListaVacia ) {
41             excepcionListaVacia.printStackTrace();
42          }
43       }
44
45    } // fin de la clase PruebaCola
```

```
La cola es: true

La cola es: true $

La cola es: true $ 34567

La cola es: true $ 34567 hola

true se quito de la cola
La cola es: $ 34567 hola

$ se quito de la cola
La cola es: 34567 hola

34567 se quito de la cola
La cola es: hola

hola se quito de la cola
Cola vacia
com.deitel.cpej5.cap20.ExcepcionListaVacia: cola esta vacia
        at com.deitel.cpej5.cap20.Lista.eliminarDelFrente(Lista.java:82)
        at com.deitel.cpej5.cap20.Cola.dequeue(Cola.java:23)
        at PruebaCola.main(PruebaCola.java:33)
```

Figura 20.14 Programa para procesar una cola. (Parte 2 de 2.)

20.7 Árboles

Las listas enlazadas, pilas y colas son *estructuras de datos lineales* (es decir, *secuencias*). Un árbol es una estructura de datos bidimensional no lineal, con propiedades especiales. Los nodos de un árbol contienen dos o más enlaces. En esta sección hablaremos sobre los *árboles binarios* (figura 20.15): los árboles cuyos nodos contienen dos enlaces (uno de los cuales puede ser `null`). El *nodo raíz* es el primer nodo en un árbol. Cada enlace en el nodo raíz hace referencia a un *hijo*. El *hijo izquierdo* es el primer nodo en el *subárbol izquierdo* (también conocido como el nodo raíz del subárbol izquierdo). El *hijo derecho* es el primer nodo en el *subárbol derecho* (también conocido como el nodo raíz del subárbol derecho). Los hijos de un nodo específico se llaman *hermanos*. Un nodo sin hijos se llama *nodo hoja*. Generalmente, los científicos computacionales dibujan árboles desde el nodo raíz hacia abajo; exactamente lo opuesto a la manera en que crecen los árboles naturales.

En nuestro ejemplo de árbol binario crearemos un árbol binario especial, conocido como *árbol de búsqueda binaria*. Un árbol de búsqueda binaria (sin valores de nodo duplicados) cuenta con la característica de que los valores en cualquier subárbol izquierdo son menores que el valor del nodo padre de ese subárbol, y los valores en cualquier subárbol derecho son mayores que el valor del nodo padre de ese subárbol. En la figura 20.16 se muestra un árbol de búsqueda binaria con 12 valores enteros. Observe que la forma del árbol de búsqueda binaria que corresponde a un conjunto de datos puede variar, dependiendo del orden en el que se inserten los valores en el árbol.

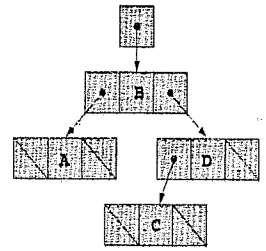


Figura 20.15 Representación gráfica de un árbol binario.

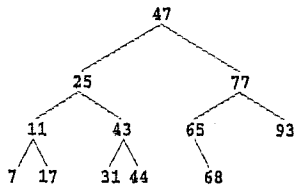


Figura. 20.16 Árbol de búsqueda binaria que contiene 12 valores.

La aplicación de las figuras 20.17 y 20.18 crea un árbol de búsqueda binaria compuesto por valores enteros, y lo recorre (es decir, avanza a través de todos sus nodos) de tres maneras: usando los *recorridos inorden*, *preorden* y *postorden*. El programa genera 10 números aleatorios e inserta a cada uno de ellos en el árbol. La clase `Arbol` se declara en el paquete `com.deitel.cpej5.cap20` para fines de reutilización.

```
1  // Fig. 20.17: Arbol.java
2  // Definición de las clases NodoArbol y Arbol.
3  package com.deitel.cpej5.cap20;
4
5  // definición de la clase NodoArbol
6  class NodoArbol {
7
8     // miembros de acceso del paquete
9     NodoArbol nodoIzquierdo;
10    int datos;
11    NodoArbol nodoDerecho;
12
13    // inicializar datos y hacer de este nodo un nodo hoja
14    public NodoArbol( int datosNodo )
15    {
16       datos = datosNodo;
17       nodoIzquierdo = nodoDerecho = null;  // el nodo no tiene hijos
18    }
19
20    // localizar punto de inserción e insertar nuevo nodo; ignorar valores duplicados
21    public synchronized void insertar( int valorInsertar )
```

Figura 20.17 Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 1 de 3.)

http://librosysoluciones.net

```
22     {
23        // insertar en subárbol izquierdo
24        if ( valorInsertar < datos ) {
25
26           // insertar nuevo NodoArbol
27           if ( nodoIzquierdo == null )
28              nodoIzquierdo = new NodoArbol( valorInsertar );
29
30           else // continuar recorriendo subárbol izquierdo
31              nodoIzquierdo.insertar( valorInsertar );
32        }
33
34        // insertar en subárbol derecho
35        else if ( valorInsertar > datos ) {
36
37           // insertar nuevo NodoArbol
38           if ( nodoDerecho == null )
39              nodoDerecho = new NodoArbol( valorInsertar );
40
41           else // continuar recorriendo subárbol derecho
42              nodoDerecho.insertar( valorInsertar );
43        }
44
45     } // fin del método insertar
46
47  } // fin de la clase NodoArbol
48
49  // definición de la clase Arbol
50  public class Arbol {
51     private NodoArbol raiz;
52
53     // construir un objeto Arbol vacío de enteros
54     public Arbol()
55     {
56        raiz = null;
57     }
58
59     // insertar un nuevo nodo en el árbol de búsqueda binaria
60     public synchronized void insertarNodo( int valorInsertar )
61     {
62        if ( raiz == null )
63           raiz = new NodoArbol( valorInsertar ); // crear el nodo raiz aquí
64
65        else
66           raiz.insertar( valorInsertar ); // llamar al método insertar
67     }
68
69     // empezar recorrido preorden
70     public synchronized void recorridoPreorden()
71     {
72        ayudantePreorden( raiz );
73     }
74
75     // método recursivo para realizar recorrido preorden
```

Figura 20.17 Declaraciones de las clases **NodoArbol** y **Arbol** para un árbol de búsqueda binaria. (Parte 2 de 3.)

http://librosysolucionarios.net


```
76     private void ayudantePreorden( NodoArbol nodo )
77     {
78        if ( nodo == null )
79           return;
80
81        System.out.print( nodo.datos + " " );   // mostrar datos del nodo
82        ayudantePreorden( nodo.nodoIzquierdo );  // recorrer subárbol izquierdo
83        ayudantePreorden( nodo.nodoDerecho );    // recorrer subárbol derecho
84     }
85
86     // empezar recorrido inorden
87     public synchronized void recorridoInorden()
88     {
89        ayudanteInorden( raiz );
90     }
91
92     // método recursivo para realizar recorrido inorden
93     private void ayudanteInorden( NodoArbol nodo )
94     {
95        if ( nodo == null )
96           return;
97
98        ayudanteInorden( nodo.nodoIzquierdo );   // recorrer subárbol izquierdo
99        System.out.print( nodo.datos + " " );    // mostrar datos del nodo
100       ayudanteInorden( nodo.nodoDerecho );     // recorrer subárbol derecho
101    }
102
103    // empezar recorrido postorden
104    public synchronized void recorridoPostorden()
105    {
106       ayudantePostorden( raiz );
107    }
108
109    // método recursivo para realizar recorrido postorden
110    private void ayudantePostorden( NodoArbol nodo )
111    {
112       if ( nodo == null )
113          return;
114
115       ayudantePostorden( nodo.nodoIzquierdo );  // recorrer subárbol izquierdo
116       ayudantePostorden( nodo.nodoDerecho );    // recorrer subárbol derecho
117       System.out.print( nodo.datos + " " );     // mostrar datos del nodo
118    }
119
120 } // fin de la clase Arbol
```

Figura 20.17 Declaraciones de las clases **NodoArbol** y **Arbol** para un árbol de búsqueda binaria. (Parte 3 de 3.)

Analicemos el programa del árbol binario. El método **main** de la clase **PruebaArbol** (figura 20.18) empieza creando una instancia de un objeto **Arbol** vacío y asigna su referencia a la variable **arbol** (línea 9). En las líneas 15 a 19 se generan 10 enteros al azar, cada uno de los cuales se inserta en el árbol binario mediante una llamada al método **synchronized insertarNodo** (línea 18). Después el programa realiza recorridos a través del árbol, inorden y postorden (los cuales explicaremos en breve) de **arbol** (líneas 22, 25 y 28, respectivamente).

```
1  // Fig. 20.18: PruebaArbol.java
2  // Este programa prueba la clase Arbol.
3  import com.deitel.cpej5.cap20.Arbol;
4
5  public class PruebaArbol {
6
7     public static void main( String args[] )
8     {
9        Arbol arbol = new Arbol();
10       int valor;
11
12       System.out.println( "Insertando los siguientes valores: " );
13
14       // insertar 10 enteros aleatorios del 0 al 99 en arbol
15       for ( int i = 1; i <= 10; i++ ) {
16          valor = ( int ) ( Math.random() * 100 );
17          System.out.print( valor + " " );
18          arbol.insertarNodo( valor );
19       }
20
21       System.out.println ( "\n\nRecorrido preorden" );
22       arbol.recorridoPreorden(); // realizar recorrido preorden de arbol
23
24       System.out.println ( "\n\nRecorrido inorden" );
25       arbol.recorridoInorden(); // realizar recorrido inorden de arbol
26
27       System.out.println ( "\n\nRecorrido postorden" );
28       arbol.recorridoPostorden(); // realizar recorrido postorden de arbol
29       System.out.println();
30    }
31
32 } // fin de la clase PruebaArbol
```

```
Insertando los siguientes valores:
65 39 79 74 0 63 55 95 55 3

Recorrido preorden
65 39 0 3 63 55 79 74 95

Recorrido inorden
0 3 39 55 63 65 74 79 95

Recorrido postorden
3 0 55 63 39 74 95 79 65
```

Figura 20.18 Programa de prueba de árboles binarios.

La clase **Arbol** (figura 20.17, líneas 50 a 120) tiene un campo **private** llamado **raiz** (línea 51); una referencia tipo **NodoArbol** al nodo raíz del árbol. El constructor de **Arbol** (líneas 54 a 57) inicializa **raiz** con **null** para indicar que el árbol está vacío. La clase contiene el método **insertarNodo** (líneas 60 a 67) para insertar un nuevo nodo en el árbol, además de los métodos **recorridoPreorden** (líneas 70 a 73), **recorridoInorden** (líneas 87 a 90) y **recorridoPostorden** (líneas 104 a 107) para empezar recorridos del árbol. Cada uno de estos métodos llama a un método utilitario recursivo para realizar la operación de recorrido en la representación interna del árbol.

http://librosysolucionarios.net

El método **synchronized insertarNodo** de la clase **Arbol** (líneas 60 a 67) determina primero si el árbol está vacío. De ser así, en la línea 63 se asigna un nuevo objeto **NodoArbol**, se inicializa el nodo con el entero que se va a insertar en el árbol y se asigna el nuevo nodo a la referencia **raiz**. Si el árbol no está vacío, en la línea 66 se hace una llamada al método **insertar** de **NodoArbol** (líneas 21 a 45). Este método utiliza la recursividad para determinar la posición del nuevo nodo en el árbol e inserta el nodo en esa posición. *En un árbol de búsqueda binaria, un nodo puede insertarse solamente como nodo hoja.*

El método **insertar** de **NodoArbol** compara el valor a insertar con el valor de **datos** en el nodo raíz. Si el valor a insertar es menor que los datos del nodo raíz (línea 24), el programa determina si el subárbol izquierdo está vacío (línea 27). De ser así, en la línea 28 se asigna un nuevo objeto **NodoArbol**, se inicializa con el entero que se va a insertar y se asigna el nuevo nodo a la referencia **nodoIzquierdo**. En caso contrario, en la línea 31 se hace una llamada recursiva a **insertar** para que se inserte el valor en el subárbol izquierdo. Si el valor a insertar es mayor que los datos del nodo raíz (línea 35), el programa determina si el subárbol derecho está vacío (línea 38). De ser así, en la línea 39 se asigna un nuevo objeto **NodoArbol**, se inicializa con el entero que se va a insertar y se asigna el nuevo nodo a la referencia **nodoDerecho**. En caso contrario, en la línea 42 se hace una llamada recursiva a **insertar** para que se inserte el valor en el subárbol derecho. Si el **valorInsertar** ya se encuentra en el árbol, simplemente se ignora.

Los métodos **recorridoInorden**, **recorridoPreorden** y **recorridoPostorden** llaman a los métodos ayudantes de **Arbol** llamados **ayudanteEnorden** (líneas 93 a 101), **ayudantePreorden** (líneas 76 a 84) y **ayudantePostorden** (líneas 110 a 118), respectivamente, para recorrer el árbol e imprimir los valores de los nodos. Los métodos ayudantes en la clase **Arbol** permiten al programador iniciar un recorrido sin tener que pasar el nodo **raiz** al método. La referencia **raiz** es un detalle de implementación que no debe ser accesible para el programador. Los métodos **recorridoInorden**, **recorridoPreorden** y **recorridoPostorden** simplemente toman la referencia privada **raiz** y la pasan al método ayudante apropiado para iniciar un recorrido del árbol. El caso base para cada método ayudante determina si la referencia que recibe es **null** y, de ser así, regresa inmediatamente.

El método **ayudanteInorden** (líneas 93 a 101) define los pasos para un recorrido inorden:

1. Recorrer el subárbol izquierdo con una llamada a **ayudanteInorden** (línea 98).
2. Procesar el valor en el nodo (línea 99).
3. Recorrer el subárbol derecho con una llamada a **ayudanteInorden** (línea 100).

El recorrido inorden no procesa el valor en un nodo sino hasta que se procesan los valores en el subárbol izquierdo de ese nodo. El recorrido inorden del árbol de la figura 20.19 es:

```
6 13 17 27 33 42 48
```

Observe que el recorrido inorden de un árbol de búsqueda binaria imprime los valores de los nodos en orden ascendente. El proceso de crear un árbol de búsqueda binaria ordena los datos de antemano; por lo tanto, a este proceso se le conoce como *ordenamiento de árbol binario*.

El método **ayudantePreorden** (líneas 76 a 84) define los pasos para un recorrido preorden:

1. Procesar el valor en el nodo (línea 81).
2. Recorrer el subárbol izquierdo con una llamada a **ayudantePreorden** (línea 82).
3. Recorrer el subárbol derecho con una llamada a **ayudantePreorden** (línea 83).

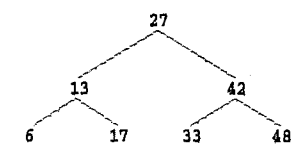


Figura 20.19 Árbol de búsqueda binaria con siete valores.

El recorrido preorden procesa el valor en cada uno de los nodos, a medida que se van visitando. Después de procesar el valor en un nodo dado, el recorrido preorden procesa los valores en el subárbol izquierdo y después los valores en el subárbol derecho. El recorrido preorden del árbol de la figura 20.19 es:

```
27 13 6 17 42 33 48
```

El método `ayudantePostorden` (líneas 110 a 118) define los pasos para un recorrido postorden:

1. Recorrer el subárbol izquierdo con una llamada a `ayudantePostorden` (línea 115).
2. Recorrer el subárbol derecho con una llamada a `ayudantePostorden` (línea 116).
3. Procesar el valor en el nodo (línea 117).

El recorrido postorden procesa el valor en cada nodo después de procesar los valores de todos los hijos de ese nodo. El `recorridoPostorden` del árbol de la figura 20.19 es:

```
6 17 13 33 48 42 27
```

El árbol de búsqueda binaria facilita la *eliminación de valores duplicados*. Al crear un árbol, la operación de inserción reconoce los intentos de insertar un valor duplicado, ya que éste sigue las mismas decisiones de "ir a la izquierda" o "ir a la derecha" en cada comparación, al igual que el valor original. Por lo tanto, la operación de inserción eventualmente comparará el valor duplicado con un nodo que contenga el mismo valor. En este punto, la operación de inserción puede decidir descartar el valor duplicado (como lo hicimos en este ejemplo).

Buscar en un árbol binario un valor que concuerde con una clave es un proceso rápido, especialmente para los árboles *estrechamente empaquetados* (o *balanceados*). En un árbol estrechamente empaquetado, cada nivel contiene cerca del doble de elementos que el nivel anterior. La figura 20.19 es un árbol binario estrechamente empaquetado. Un árbol de búsqueda binaria estrechamente empaquetado con n elementos tiene $\log_2 n$ niveles. Por lo tanto, se requieren cuando mucho $\log_2 n$ comparaciones para encontrar una concordancia o determinar que no existe una. La búsqueda en un árbol de búsqueda binaria de 1000 elementos (estrechamente empaquetado) requiere cuando mucho de 10 comparaciones, ya que $2^{10} > 1000$. La búsqueda en un árbol de búsqueda binaria de 1,000,000 de elementos (estrechamente empaquetado) requiere cuando mucho de 20 comparaciones, ya que $2^{20} > 1{,}000{,}000$.

Los ejercicios de este capítulo presentan algoritmos para varias operaciones más de árboles binarios, como eliminar un elemento de un árbol binario, imprimir un árbol binario en formato de árbol bidimensional y realizar un *recorrido en orden de niveles de un árbol binario*. En el recorrido en orden de niveles de un árbol binario se visitan sus nodos fila por fila, empezando en el nivel del nodo raíz. En cada nivel del árbol, un recorrido en orden de niveles visita los nodos de izquierda a derecha. Otros ejercicios de árboles binarios incluyen el permitir que un árbol de búsqueda binaria contenga valores duplicados, insertar valores de cadena en un árbol binario y determinar cuántos niveles hay en un árbol binario. En los capítulos 21 y 22 continuaremos con nuestra discusión sobre las estructuras de datos, al presentar las estructuras de datos incluidas en la API de Java.

RESUMEN

- Las estructuras de datos dinámicas pueden crecer y reducirse en tiempo de ejecución.
- Las listas enlazadas son colecciones de elementos de datos "alineados en una fila"; pueden insertarse y eliminarse elementos en cualquier parte de una lista enlazada.
- Las pilas son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos solamente en un extremo de una pila: su *parte superior*.
- Las colas representan líneas de espera; se insertan elementos en la parte final y se eliminan de su parte inicial.
- Los árboles binarios facilitan la búsqueda y ordenamiento de los datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos y la compilación de expresiones en lenguaje máquina.
- Una clase autorreferenciada contiene una referencia a otro objeto del mismo tipo de clase. Los objetos autorreferenciados pueden enlazarse entre sí para formar estructuras de datos dinámicas.

- El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de espacio en disco disponible en un sistema con memoria virtual. A menudo, los límites son mucho más pequeños ya que la memoria disponible de la computadora debe compartirse entre muchos usuarios.
- Si no hay memoria disponible, se lanza una excepción `OutOfMemoryError`.
- Una lista enlazada se utiliza mediante una referencia al primer nodo de la lista. Cada nodo subsiguiente se utiliza a través del miembro de referencia de enlace almacenado en el nodo anterior.
- Por convención, la referencia de enlace en el último nodo de una lista se establece en `null` para indicar el final de la lista.
- Un nodo puede contener datos de cualquier tipo, incluyendo objetos de otras clases.
- Una lista enlazada es apropiada cuando el número de elementos de datos que se van a almacenar es impredecible. Las listas enlazadas son dinámicas, por lo que su longitud puede incrementarse o reducirse, según sea necesario.
- El tamaño de un arreglo "convencional" en Java no puede alterarse; el tamaño se fija al momento de su creación.
- Las listas enlazadas pueden mantenerse en orden con sólo insertar cada nuevo elemento en el punto apropiado de la lista.
- Generalmente, los nodos de las listas no se almacenan de manera contigua en memoria. En vez de ello, son adyacentes en forma lógica.
- Los métodos que manipulan el contenido de una lista deben declararse como `synchronized`, para que los objetos de la lista puedan usarse con *seguridad en los subprocesos* de un programa con subprocesamiento múltiple. Si un subproceso está modificando el contenido de una lista, ningún otro subproceso podrá modificar la misma lista al mismo tiempo.
- A una pila se le conoce como estructura de datos UEPS (último en entrar, primero en salir). Los métodos principales usados para manipular una pila son empujar (`push`) y sacar (`pop`). El método `push` agrega un nuevo nodo a la parte superior de la pila. El método `pop` elimina un nodo de la parte superior de la pila y devuelve el objeto `datos` del nodo que se quitó.
- Las pilas tienen muchas aplicaciones interesantes. Cuando se hace la llamada a un método, el método llamado debe saber cómo regresar a su invocador, por lo que la dirección de retorno se mete en la pila de ejecución del programa. Si ocurre una serie de llamadas a métodos, los valores de retorno sucesivos se meten en la pila en el orden "último en entrar, primero en salir", para que cada método pueda regresar a su invocador. La pila de ejecución del programa contiene el espacio creado para las variables locales en cada invocación de un método. Cuando el método regresa a su invocador, el espacio para las variables locales de ese método se saca de la pila y esas variables dejan de ser conocidas por el programa.
- Los compiladores utilizan pilas para evaluar expresiones aritméticas y generar código en lenguaje máquina para procesar las expresiones.
- La técnica de implementar cada método de la pila como una llamada a un método de `Lista` se conoce como delegación; el método invocado de la pila delega la llamada al método apropiado de `Lista`.
- Una cola es similar a la línea para pagar en un supermercado: la primera persona en la línea es atendida primero, y los demás clientes entran a la línea sólo por su parte final, esperando a ser atendidos.
- Los nodos de una cola se eliminan sólo desde el principio de la misma y se insertan sólo al final de ésta. Por esta razón, a una cola se le conoce como estructura de datos PEPS (primero en entrar, primero en salir).
- Las operaciones para insertar y eliminar en una cola se conocen como `enqueue` (agregar a la cola) y `dequeue` (retirar de la cola).
- Las colas tienen muchas aplicaciones en los sistemas computacionales. La mayoría de las computadoras tienen sólo un procesador, por lo que sólo pueden atender a un usuario a la vez. Las entradas para los otros usuarios se colocan en una cola. La entrada al frente de la cola es la siguiente que recibe atención. Cada entrada avanza gradualmente al frente de la cola, a medida que los usuarios reciben atención.
- Un árbol es una estructura de datos bidimensional, no lineal. Los nodos de un árbol contienen dos o más enlaces.
- Un árbol binario es un árbol cuyos nodos contienen dos enlaces. El nodo raíz es el primer nodo en un árbol.
- Cada enlace en el nodo raíz hace referencia a un *hijo*. El hijo izquierdo es el primer nodo en el subárbol izquierdo, y el hijo derecho es el primer nodo en el subárbol derecho.
- Los hijos de un nodo se llaman hermanos. Un nodo sin hijos se llama nodo hoja.
- Un árbol de búsqueda binaria (sin valores de nodo duplicados) cuenta con la característica de que los valores en cualquier subárbol izquierdo son menores que el valor en su nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en su nodo padre. En un árbol de búsqueda binaria, un nodo puede insertarse solamente como nodo hoja.
- El recorrido inorden de un árbol de búsqueda binaria procesa los valores de los nodos en orden ascendente.

http://librosysolucionarios.net

- En un recorrido preorden, el valor en cada uno de los nodos se procesa a medida que se van visitando. Después se procesan los valores en el subárbol izquierdo y, por último, los valores en el subárbol derecho.
- En un recorrido postorden, el valor en cada uno de los nodos se procesa después de los valores de sus hijos.
- El árbol de búsqueda binaria facilita la eliminación de valores duplicados. Al crear un árbol se reconocen los intentos de insertar un valor duplicado, ya que éste sigue las mismas decisiones de "ir a la izquierda" o "ir a la derecha" en cada comparación, al igual que el valor original. Por lo tanto, eventualmente se compara el valor duplicado con un nodo que contenga el mismo valor. El valor duplicado puede descartarse en este punto.
- Buscar en un árbol binario un valor que concuerde con una clave es también un proceso rápido, en especial para los árboles estrechamente empaquetados. En un árbol estrechamente empaquetado, cada nivel contiene cerca del doble de elementos que el nivel anterior. Por lo tanto, un árbol de búsqueda binaria estrechamente empaquetado con n elementos tiene $\log_2 n$ niveles, por lo que tendrían que hacerse cuando mucho $\log_2 n$ comparaciones para encontrar una concordancia o determinar que no existe una. La búsqueda en un árbol de búsqueda binaria de 1000 elementos (estrechamente empaquetado) requiere cuando mucho de 10 comparaciones, ya que $2^{10} > 1000$. La búsqueda en un árbol de búsqueda binaria de 1,000,000 de elementos (estrechamente empaquetado) requiere cuando mucho de 20 comparaciones, ya que $2^{20} > 1{,}000{,}000$.

TERMINOLOGÍA

agregar a la cola (enqueue)
algoritmo recursivo para recorrer árboles
árbol
árbol binario
árbol de búsqueda binaria
árboles balanceados
árboles estrechamente empaquetados
clase autorreferenciada
cola
delegar la llamada a un método
eliminación de valores duplicados
eliminar un nodo
estructura de datos dinámica
estructuras de datos lineales
estructuras de datos no lineales
hijo derecho
hijo izquierdo
hijos de un nodo
insertar un nodo
lista enlazada
método predicado
nodo
nodo hijo
nodo hoja
nodo padre
nodo raíz
`null`, referencia
ordenamiento de árbol binario
`OutOfMemoryError`
parte final de una cola
parte inicial de una cola
parte superior de una pila
PEPS (primero en entrar, primero en salir)
pila
pila de ejecución del programa
recorrido
recorrido en orden de niveles de un árbol binario
recorrido inorden de un árbol binario
recorrido postorden de un árbol binario
recorrido preorden de un árbol binario
retirar de la cola (dequeue)
sacar (pop)
subárbol
subárbol derecho
subárbol izquierdo
UEPS (último en entrar, primero en salir)
visitar un nodo

EJERCICIOS DE AUTOEVALUACIÓN

20.1 Llene los espacios en blanco en cada uno de los siguientes enunciados:

a) Una clase _____ se utiliza para formar estructuras de datos dinámicas que pueden crecer y reducirse en tiempo de ejecución.
b) Una _____ es una versión restringida de una lista enlazada, en la que pueden insertarse y eliminarse nodos solamente desde el principio de la lista.
c) Un método que no altera una lista enlazada, sino que sólo la analiza para determinar si está vacía, se conoce como método _____.
d) A una cola se le conoce como estructura de datos _____, ya que los primeros nodos que se insertan son los primeros que se eliminan.
e) La referencia al siguiente nodo en una lista enlazada se conoce como un _____.
f) Al proceso de reclamar automáticamente la memoria asignada en forma dinámica se le conoce como _____.
g) Una _____ es una versión restringida de una lista enlazada, en la que pueden insertarse nodos al final de la lista y eliminarse solamente desde el principio.
h) Un _____ es una estructura de datos bidimensional no lineal, que contiene nodos con dos o más enlaces.

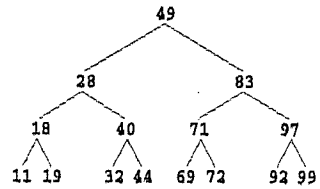


Figura 20.20 Árbol de búsqueda binaria con 15 nodos.

i) A una pila se le conoce como estructura de datos _____, ya que el último nodo insertado es el primero que se elimina.
j) Los nodos de un árbol _____ contienen dos miembros de enlace.
k) El primer nodo de un árbol es el nodo _____.
l) Cada enlace en el nodo de un árbol hace referencia a un _____ o _____ de ese nodo.
m) El nodo de un árbol que no tiene hijos se llama nodo _____.
n) Los tres algoritmos de recorrido que mencionamos en el texto para los árboles de búsqueda binaria son _____, _____ y _____.

20.2 ¿Cuáles son las diferencias entre una lista enlazada y una pila?

20.3 ¿Cuáles son las diferencias entre una pila y una cola?

20.4 Tal vez un título más apropiado para este capítulo hubiera sido "Estructuras de datos reutilizables". Escriba sus comentarios acerca de cómo contribuyen cada una de las siguientes entidades o conceptos a la reutilización de las estructuras de datos:

a) clases
b) herencia
c) composición

20.5 Proporcione manualmente los recorridos inorden, preorden y postorden del árbol de búsqueda binaria de la figura 20.20.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

20.1 a) Autorreferenciada. b) Pila. c) Predicado. d) PEPS (primero en entrar, primero en salir). e) Enlace. f) Recolección de basura. g) Cola. h) árbol. i) UEPS (último en entrar, primero en salir). j) Binario. k) Raíz. l) Hijo o subárbol. m) Hoja. n) Inorden, preorden, postorden.

20.2 Es posible insertar y eliminar un nodo en cualquier lugar de una lista enlazada. Los nodos en una pila pueden insertarse solamente en la parte superior y eliminarse desde la parte superior de una pila.

20.3 Una cola es una estructura de datos PEPS que tiene referencias tanto a su parte inicial como a su parte final, de manera que pueden insertarse nodos al final y eliminarse del principio de la cola. Una pila es una estructura de datos UEPS que tiene una sola referencia a la parte superior de la pila, en donde se llevan a cabo las operaciones de inserción y eliminación de nodos.

20.4 a) Las clases nos permiten crear tantas instancias de todos los objetos de estructura de datos de cierto tipo (es decir, clase) como sea necesario.
b) La herencia permite a una subclase reutilizar la funcionalidad de una superclase. Los métodos públicos de una superclase pueden utilizarse a través de una subclase, para eliminar la lógica duplicada.
c) La composición permite a una clase reutilizar código almacenando una instancia de otra clase en un campo. Los métodos públicos de la clase miembro pueden ser llamados por los métodos de la clase compuesta.

20.5 El recorrido inorden es:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

El recorrido preorden es:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

El recorrido postorden es:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

http://librosysolucionarios.net

EJERCICIOS

20.6 Escriba un programa para concatenar dos objetos de lista enlazada de caracteres. La clase `ConcatenarLista` debe incluir un método llamado `concatenar` que tome referencias a ambos objetos lista como argumentos y que concatene la segunda lista con la primera.

20.7 Escriba un programa para fusionar dos objetos de lista ordenada de enteros en un solo objeto de lista ordenada de enteros. El método `fusionar` de la clase `FusionaLista` debe recibir referencias a cada uno de los objetos lista que se van a fusionar, y debe devolver una referencia al objeto lista fusionado.

20.8 Escriba un programa para insertar 25 números aleatorios de 0 a 100 en orden, en un objeto lista enlazada. El programa deberá calcular la suma de los elementos y el promedio de punto flotante de los elementos.

20.9 Escriba un programa para crear un objeto lista enlazada de 10 caracteres, y que luego cree un segundo objeto lista que contenga una copia de la primera lista, pero en orden inverso.

20.10 Escriba un programa que reciba una línea de texto como entrada y que utilice un objeto pila para imprimir las palabras de la línea en orden inverso.

20.11 Escriba un programa que utilice una pila para determinar si una cadena es un palíndromo (es decir, que la cadena se deletree en forma idéntica, tanto al revés como al derecho). El programa debe ignorar espacios y puntuación.

20.12 Los compiladores utilizan pilas para ayudar en el proceso de evaluar expresiones y generar código en lenguaje máquina. En este ejercicio y en el siguiente, investigaremos cómo los compiladores evalúan expresiones aritméticas que consisten solamente de constantes, operadores y paréntesis.

Los humanos generalmente escriben expresiones como `3 + 4` y `7 / 9`, en donde el operador (`+` o `/` aquí) se escribe entre sus operandos; a esta notación se le conoce como *notación infijo*. Las computadoras "prefieren" la *notación postfijo*, en donde el operador se escribe a la derecha de sus dos operandos. Las anteriores expresiones infijo aparecerían en notación postfijo como `3 4 +` y `7 9 /`, respectivamente.

Para evaluar una expresión infijo compleja, un compilador primero convertiría la expresión en notación postfijo y evaluaría la versión postfijo de la expresión. Cada uno de estos algoritmos requiere solamente de una pasada de izquierda a derecha de la expresión. Cada algoritmo utiliza un objeto pila para dar soporte a su operación y, en cada algoritmo, la pila se utiliza para un propósito distinto.

En este ejercicio, usted escribirá una versión en Java del algoritmo de conversión infijo a postfijo. En el siguiente ejercicio, usted escribirá una versión en Java del algoritmo de evaluación de expresiones postfijo. En un ejercicio posterior, descubrirá que el código que escriba en este ejercicio podrá ayudarle a implementar un compilador completamente funcional.

Escriba la clase `ConvertidorInfijoAPostfijo` para convertir una expresión aritmética infijo ordinaria (suponga que se escribe una expresión válida) con enteros de un solo dígito, como:

```
(6 + 2) * 5 - 8 / 4
```

en una expresión postfijo. La versión postfijo de la expresión infijo anterior es (observe que no se necesitan paréntesis):

```
6 2 + 5 * 8 4 / -
```

El programa debe leer la expresión y colocarla en la variable `StringBuffer infijo`, y utilizar una de las clases de pila implementadas en este capítulo para ayudar a crear la expresión postfijo en la variable `StringBuffer postfijo`. El algoritmo para crear una expresión postfijo es el siguiente:

a) Meter un paréntesis izquierdo `'('` en la pila.
b) Anexar un paréntesis derecho `')'` al final de `infijo`.
c) Mientras que la pila no esté vacía, leer `infijo` de izquierda a derecha y hacer lo siguiente:
   Si el carácter actual en `infijo` es un dígito, anexarlo a `postfijo`.
   Si el carácter actual en `infijo` es un paréntesis izquierdo, meterlo a la pila.
   Si el carácter actual en `infijo` es un operador:
      Sacar los operadores (si los hay) de la parte superior de la pila, mientras tengan igual o mayor precedencia que el operador actual, y anexar los operadores que se sacaron a `postfijo`.
      Meter en la pila el carácter actual de `infijo`.
   Si el carácter actual en `infijo` es un paréntesis derecho:
      Sacar operadores de la parte superior de la pila y anexarlos a `postfijo`, hasta que haya un paréntesis izquierdo en la parte superior de la pila.
      Sacar (y descartar) el paréntesis izquierdo de la pila.

Las siguientes operaciones aritméticas se permiten en una expresión:

`+` suma
`-` resta
`*` multiplicación
`/` división
`^` exponenciación
`%` residuo

La pila debe mantenerse con nodos de pila que contengan, cada uno, una variable de instancia y una referencia al siguiente nodo de la pila. Algunos de los métodos que puede proporcionar son:

a) El método `convertirAPostfijo`, que convierte la expresión infijo a notación postfijo.
b) El método `esOperador`, el cual determina si `c` es un operador.
c) El método `precedencia`, que determina si la precedencia de `operador1` (de la expresión infijo) es menor, igual o mayor que la precedencia de `operador2` (de la pila). El método devuelve `true` si `operador1` tiene menor precedencia que `operador2`. En caso contrario, se devuelve `false`.
d) El método `parteSuperiorPila` (éste debe agregarse a la clase pila), que devuelve el valor de la parte superior de la pila sin sacarlo de la misma.

20.13 Escriba la clase `EvaluadorPostfijo`, el cual evalúa una expresión postfijo como:

```
6 2 + 5 * 8 4 / -
```

El programa debe leer una expresión postfijo que consista de dígitos y operadores, para después colocarla en un objeto `StringBuffer`. Utilizando versiones modificadas de los métodos de pila implementados anteriormente en este capítulo, el programa deberá explorar la expresión y evaluarla (suponiendo que sea válida). El algoritmo es el siguiente:

a) Anexar un paréntesis derecho `')'` al final de la expresión postfijo. Cuando se encuentre el carácter de paréntesis derecho, ya no habrá nada más qué procesar.
b) Cuando no se encuentre el carácter de paréntesis derecho, leer la expresión de izquierda a derecha.
   Si el carácter actual es un dígito, hacer lo siguiente:
   Meter su valor entero en la pila (el valor entero de un carácter tipo dígito es su valor en el conjunto de caracteres de la computadora menos el valor de `'0'` en Unicode).
   En caso contrario, si el carácter actual es un *operador*:
   Sacar los dos elementos superiores de la pila y colocarlos en las variables `x` y `y`.
   Calcular `y` *operador* `x`.
   Meter el resultado del cálculo en la pila.
c) Cuando se encuentre el paréntesis derecho en la expresión, sacar el valor superior de la pila. Éste es el resultado de la expresión postfijo.

[*Nota:* En el inciso b) anterior (con base en la expresión de ejemplo al principio de los ejercicios), si el operador es `'/'`, el valor superior de la pila es `2` y el siguiente elemento en la pila es `8`, entonces sacar `2` y colocarlo en `x`, sacar `8` y colocarlo en `y`, evaluar `8 / 2` y meter el resultado (`4`) devuelta en la pila. Esta nota también se aplica al operador `'-'`.] Las operaciones aritméticas permitidas en una expresión son:

`+` suma
`-` resta
`*` multiplicación
`/` división
`^` exponenciación
`%` residuo

La pila debe mantenerse con una de las clases de pila que se presentaron en este capítulo. Tal vez usted pueda proporcionar los siguientes métodos:

a) El método `evaluarExpresionPostfijo`, el cual evalúa la expresión postfijo.
b) El método `calcular`, el cual evalúa la expresión `op1 operador op2`.
c) El método `push`, que mete un valor en la pila.
d) El método `pop`, que saca un valor de la pila.
e) El método `estaVacia`, que determina si la pila está vacía.
f) El método `imprimirPila`, el cual imprime la pila.

20.14 Modifique el programa evaluador de expresiones postfijo del ejercicio 20.13, de manera que pueda procesar operandos enteros mayores que 9.

http://librosysolucionarios.net

20.15 (*Simulación de supermercado.*) Escriba un programa que simule una línea para pagar en un supermercado. La línea es un objeto cola. Los clientes (es decir, los objetos cliente) llegan en intervalos enteros aleatorios de 1 a 4 minutos. Además, a cada cliente se le atiende en intervalos enteros aleatorios de 1 a 4 minutos. Obviamente, los ritmos necesitan balancearse. Si el ritmo promedio de llegadas es mayor que el ritmo promedio de atención, la cola crecerá infinitamente. Incluso con ritmos "balanceados", el factor aleatorio puede aún provocar largas líneas. Ejecute la simulación del supermercado durante un día de 12 horas (720 minutos), utilizando el siguiente algoritmo:

a) Elegir un entero aleatorio entre 1 y 4 para determinar el minuto en el que debe llegar el primer cliente.
b) Al momento en que llegue el cliente:
   Determinar el tiempo de atención del cliente (entero aleatorio de 1 a 4).
   Empezar a atender al cliente.
   Programar la hora de llegada del siguiente cliente (se suma un entero aleatorio de 1 a 4 al tiempo actual).
c) Para cada minuto del día:
   Si llega el siguiente cliente, hay que proceder de la siguiente manera:
      Decirlo así.
      Poner al cliente en la cola.
      Programar la hora de llegada del siguiente cliente.
   Si se terminó de atender al último cliente:
      Decirlo así.
      Sacar de la cola al siguiente cliente al que se va a atender.
      Determinar el tiempo requerido para dar servicio al cliente (se suma un entero aleatorio del 1 al 4 al tiempo actual).

Ahora ejecute su simulación durante 720 minutos y responda a cada una de las siguientes preguntas:

a) ¿Cuál es el máximo número de clientes en la cola, en cualquier momento dado?
b) ¿Cuál es el tiempo de espera más largo que experimenta un cliente?
c) ¿Qué ocurre si el intervalo de llegada se cambia de 1 a 4 minutos por un intervalo de 1 a 3 minutos?

20.16 Modifique las figuras 20.17 y 20.18 para permitir que el árbol binario contenga valores duplicados.

20.17 Escriba un programa con base en el programa de las figuras 20.17 y 20.18, que reciba como entrada una línea de texto, divida la oración en palabras separadas (tal vez quiera utilizar la clase `StreamTokenizer` del paquete `java.io`), las inserte en un árbol de búsqueda binaria e imprima los recorridos inorden, preorden y postorden del árbol.

20.18 En este capítulo vimos que la eliminación de duplicados es un proceso bastante simple cuando se crea un árbol de búsqueda binaria. Describa cómo llevaría a cabo la eliminación de duplicados utilizando sólo un arreglo unidimensional. Compare el rendimiento de la eliminación de valores duplicados con base en arreglos y el rendimiento de la eliminación de duplicados con base en árboles de búsqueda binaria.

20.19 Escriba un método llamado `profundidad` que reciba un árbol binario y determine cuántos niveles tiene.

20.20 (*Imprimir una lista en forma recursiva y en forma inversa.*) Escriba un método llamado `imprimirListaAlReves` que imprima en forma recursiva los elementos en un objeto lista enlazada, en orden inverso. Escriba un programa de prueba para crear una lista ordenada de enteros e imprimir la lista en orden inverso.

20.21 (*Buscar en una lista en forma recursiva.*) Escriba un método llamado `buscarLista` que busque en forma recursiva en un objeto lista enlazada un valor específico. El método `buscarLista` deberá devolver una referencia al valor, si es que lo encuentra; en caso contrario, deberá devolver `null`. Use su método en un programa de prueba para crear una lista de enteros. El programa deberá pedir al usuario un valor a localizar en la lista.

20.22 (*Eliminación en árboles binarios.*) En este ejercicio hablaremos sobre cómo eliminar elementos de los árboles de búsqueda binaria. El algoritmo de eliminación no es tan simple como el de inserción. Al eliminar un elemento puede haber tres casos: que el elemento esté contenido en un nodo hoja (es decir, que no tenga hijos), que esté contenido en un nodo que tenga un hijo, o que esté contenido en un nodo con dos hijos.

Si el elemento que se va a eliminar está contenido en un nodo hoja, este nodo se elimina y a la referencia en el nodo padre se le asigna el valor nulo.

Si el elemento que se va a eliminar está contenido en un nodo con un hijo, a la referencia en el nodo padre se le asigna el nodo hijo y se elimina el nodo que contenga el elemento de datos. Esto hace que el nodo hijo ocupe el lugar del nodo eliminado en el árbol.

El último caso es el más difícil. Cuando se elimina un nodo con dos hijos, otro nodo en el árbol debe tomar su lugar. Sin embargo, la referencia en el nodo padre no puede simplemente asignarse de manera que haga referencia a uno de los hijos del nodo que se va a eliminar. En la mayoría de los casos, el árbol de búsqueda binaria resultante no se adhiere a la siguiente característica de los árboles de búsqueda binaria (sin valores duplicados): *Los valores en cualquier subárbol izquierdo son menores que el valor en el nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en el nodo padre.*

¿Cuál nodo debe utilizarse como *nodo de reemplazo* para mantener esta característica? Debe ser el nodo que contenga el valor más grande en el árbol, pero que sea menor que el valor en el nodo que se va a eliminar, o el nodo que contenga el valor más pequeño en el árbol, pero que sea mayor que el valor en el nodo que se va a eliminar. Consideremos el nodo con el valor más pequeño. En un árbol de búsqueda binaria, el valor más grande que sea menor que el valor de un padre se encuentra en el subárbol izquierdo del nodo padre y se garantiza que estará contenido en el nodo que se encuentre más a la derecha del subárbol. Para localizar este nodo hay que avanzar por el subárbol izquierdo hacia abajo y a la derecha, hasta que la referencia al hijo derecho del nodo actual sea nula. Ahora estamos haciendo referencia al nodo de reemplazo, que es un nodo hoja o un nodo con un hijo a su izquierda. Si el nodo de reemplazo es un nodo hoja, los pasos para llevar a cabo la eliminación son los siguientes:

a) Almacenar la referencia al nodo que se va a eliminar en una variable de referencia temporal.
b) Hacer que la referencia en el padre del nodo que se va a eliminar haga referencia al nodo de reemplazo.
c) Asignar a la referencia en el padre del nodo de reemplazo el valor `null`.
d) Hacer que la referencia al subárbol derecho en el nodo de reemplazo haga referencia al subárbol derecho del nodo que se va a eliminar.
e) Hacer que la referencia al subárbol izquierdo en el nodo de reemplazo haga referencia al subárbol izquierdo del nodo que se va a eliminar.

Los pasos de eliminación para el caso de un nodo de reemplazo con un hijo izquierdo son similares a los pasos para un nodo de reemplazo sin hijos, sólo que el algoritmo debe también desplazar al hijo hacia la posición del nodo de reemplazo en el árbol. Si el nodo de reemplazo es un nodo con un hijo izquierdo, los pasos para llevar a cabo la eliminación son los siguientes:

a) Almacenar la referencia al nodo que se va a eliminar en una variable de referencia temporal.
b) Hacer que la referencia en el padre del nodo que se va a eliminar haga referencia al nodo de reemplazo.
c) Hacer que la referencia en el padre del nodo de reemplazo haga referencia al hijo izquierdo del nodo de reemplazo.
d) Hacer que la referencia al subárbol derecho en el nodo de reemplazo haga referencia al subárbol derecho del nodo que se va a eliminar.
e) Hacer que la referencia al subárbol izquierdo en el nodo de reemplazo haga referencia al subárbol izquierdo del nodo que se va a eliminar.

Escriba el método `eliminarNodo`, que debe tomar como argumento el valor a eliminar. El método `eliminarNodo` deberá localizar en el árbol el nodo que contenga el valor a eliminar, y deberá utilizar los algoritmos aquí descritos para eliminar el nodo. Si no se encuentra el valor en el árbol, el método deberá imprimir un mensaje que indique si el valor se eliminó. Modifique el programa de las figuras 20.17 y 20.18 para utilizar este método. Después de eliminar un elemento, llame a los métodos `recorridoInorden`, `recorridoPreorden` y `recorridoPostorden` para confirmar que la operación de eliminación se haya llevado a cabo correctamente.

20.23 (*Búsqueda en un árbol binario.*) Escriba el método `busquedaArbolBinario`, para tratar de localizar un valor especificado en un objeto árbol de búsqueda binaria. El método deberá tomar como argumento una clave de búsqueda a localizar. Si se encuentra el nodo que contenga la clave de búsqueda, el método deberá devolver una referencia a ese nodo; en caso contrario, el método deberá devolver una referencia nula.

20.24 (*Recorrido de un árbol binario en orden de niveles.*) El programa de las figuras 20.17 y 20.18 demostró el uso de tres métodos recursivos para recorrer un árbol binario: los recorridos inorden, preorden y postorden. En este ejercicio presentamos el *recorrido en orden de niveles* de un árbol binario, en el cual los valores de los nodos se imprimen nivel por nivel, empezando en el nivel del nodo raíz. Los nodos en cada nivel se imprimen de izquierda a derecha. El recorrido en orden de niveles no es un algoritmo recursivo. Utiliza un objeto cola para controlar la impresión en pantalla de los nodos. El algoritmo es el siguiente:

a) Insertar el nodo raíz en la cola.
b) Mientras haya nodos restantes en la cola:
   Obtener el siguiente nodo en la cola.
   Imprimir el valor del nodo.
   Si la referencia al hijo izquierdo del nodo no es nula:
      Insertar el nodo del hijo izquierdo en la cola.
   Si la referencia al hijo derecho del nodo no es nula:
      Insertar el nodo del hijo derecho en la cola.

Escriba el método `ordenNiveles` para llevar a cabo un recorrido en orden de niveles de un objeto árbol binario. Modifique el programa de las figuras 20.17 y 20.18 para utilizar este método. (*Nota:* También necesitará utilizar los métodos de procesamiento de colas de la figura 20.13 en este programa.)

http://librosysolucionarios.net

20.25 (*Imprimir árboles.*) Escriba un método recursivo llamado `mostrarArbol` para mostrar un objeto árbol binario en la pantalla. El método deberá mostrar el árbol fila por fila, con la parte superior del mismo a la izquierda de la pantalla y la parte inferior hacia la derecha de la pantalla. Cada fila se debe mostrar en forma vertical. Por ejemplo, el árbol binario que aparece en la figura 20.20 debe mostrarse en pantalla como se indica en la figura 20.21. Observe que el nodo hoja que se encuentra más a la derecha en el árbol aparece en la parte superior de la pantalla, en la columna que está más a la derecha, y el nodo raíz aparece a la izquierda de la pantalla. Cada columna de salida empieza cinco espacios a la derecha de la columna anterior. El método `mostrarArbol` debe recibir un argumento llamado `totalEspacios`, el cual representa el número de espacios que anteceden al valor que va a mostrarse en pantalla. (Esta variable debe empezar en cero, para que el nodo raíz se muestre a la izquierda de la pantalla.) El método utiliza un recorrido inorden modificado para mostrar el árbol en pantalla; empieza en el nodo que está más a la derecha en el árbol y avanza en retroceso hacia la izquierda. El algoritmo es el siguiente:

```
Mientras la referencia al nodo actual no sea nula:
    Llamar en forma recursiva a mostrarArbol con el subárbol derecho del nodo actual y
        totalEspacios + 5.
    Utilice una instrucción for para contar de 1 a totalEspacios e imprimir espacios.
    Mostrar el valor en el nodo actual.
    Hacer que la referencia al nodo actual haga referencia al subárbol izquierdo del nodo actual.
    Incrementar totalEspacios en 5.
```

SECCIÓN ESPECIAL: CONSTRUYA SU PROPIO COMPILADOR

En los ejercicios 7.43 y 7.44 presentamos el Lenguaje máquina Simpletron (LMS) y usted implementó un simulador de computadora Simpletron para ejecutar programas escritos en LMS. En esta sección crearemos un compilador que convierta los programas escritos en un lenguaje de alto nivel a LMS. Esta sección "enlaza" entre sí todo el proceso de programación. Usted escribirá programas en este nuevo lenguaje de alto nivel, los compilará en el compilador que va a construir y los ejecutará en el simulador que construyó en el ejercicio 7.44. Usted deberá hacer todo el esfuerzo posible por implementar su compilador con un enfoque orientado a objetos.

20.26 (*El lenguaje Simple.*) Antes de empezar a construir el compilador, hablaremos sobre un lenguaje de alto nivel simple pero poderoso, similar a las primeras versiones del popular lenguaje Basic. Llamaremos a este lenguaje *Simple*. Cada *instrucción* de Simple consiste de un *número de línea* y de una *instrucción* de Simple. Los números de línea deben aparecer en orden ascendente. Cada instrucción empieza con uno de los siguientes *comandos* de Simple: `rem`, `input`, `let`, `print`, `goto`, `if/goto` o `end` (vea la figura 20.22). Todos los comandos excepto `end` pueden utilizarse en forma repetida. Simple evalúa solamente las expresiones de enteros que utilizan los operadores `+`, `-`, `*` y `/`. Estos operadores tienen la misma precedencia que en Java. Pueden utilizarse paréntesis para cambiar el orden de evaluación de una expresión.

Nuestro compilador de Simple reconoce solamente letras en minúscula; por lo tanto, todos los caracteres en un archivo de Simple deben estar en minúsculas. (Las letras mayúsculas producen un error de sintaxis a menos que

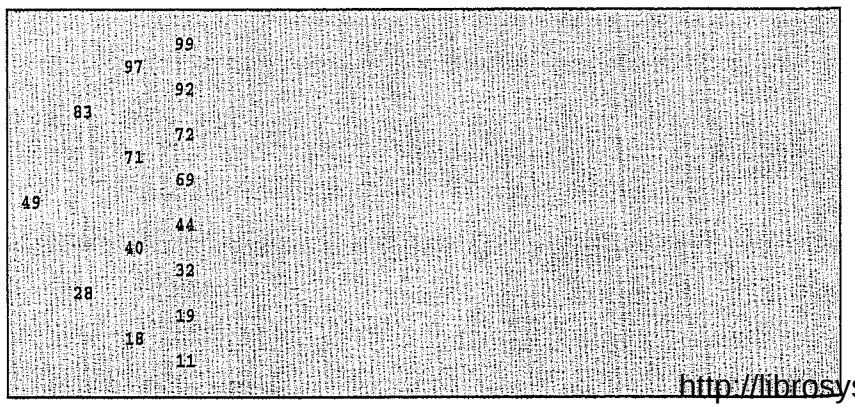


Figura 20.21 Salida de ejemplo del método recursivo `mostrarArbol`.

http://librosysolucionarios.net

Comando	Instrucción de ejemplo	Descripción
`rem`	`50 rem este es un comentario`	Cualquier texto después del comando `rem` es para fines de documentación solamente, por lo que el compilador lo ignora.
`input`	`30 input x`	Mostrar un signo de interrogación para pedir al usuario que introduzca un entero. Leer ese entero desde el teclado y almacenarlo en `x`.
`let`	`80 let u = 4 * (j - 56)`	Asignar a `u` el valor de `4 * (j - 56)`. Observe que puede aparecer una expresión arbitrariamente compleja a la derecha del signo de igual.
`print`	`10 print w`	Mostrar el valor de `w`.
`goto`	`70 goto 45`	Transferir el control del programa a la línea `45`.
`if/goto`	`35 if i == z goto 80`	Comparar si `i` y `z` son iguales y transferir el control del programa a la línea `80` si la condición es verdadera; en caso contrario, continuar la ejecución con la siguiente instrucción.
`end`	`99 end`	Terminar la ejecución del programa.

Figura 20.22 Comandos de Simple.

aparezcan en una instrucción `rem`, en cuyo caso se ignoran.) Un *nombre de variable* es una sola letra. Simple no permite el uso de nombres descriptivos para las variables, por lo que éstas deben explicarse en comentarios para indicar su uso en un programa. Simple utiliza solamente variables enteras. Simple no tiene declaraciones de variables; con sólo mencionar el nombre de una variable en un programa, ésta se declara y se inicializa con cero. La sintaxis de Simple no permite la manipulación de cadenas (leer una cadena, escribir una cadena, comparar cadenas, etcétera). Si se encuentra una cadena en un programa de Simple (después de un comando distinto de `rem`), el compilador genera un error de sintaxis. La primera versión de nuestro compilador supone que los programas de Simple se introducen correctamente. En el ejercicio 20.29 pedimos al lector que modifique el compilador para llevar a cabo la comprobación de errores de sintaxis.

Simple utiliza la instrucción `if/goto` condicional y la instrucción `goto` incondicional para alterar el flujo de control durante la ejecución del programa. Si la condición en la instrucción `if/goto` es verdadera, el control se transfiere a una línea específica del programa. Los siguientes operadores relacionales y de igualdad son válidos en una instrucción `if/goto`: `<`, `>`, `<=`, `>=`, `==` o `!=`. La precedencia de estos operadores es la misma que en Java.

Consideremos ahora varios programas para demostrar las características de Simple. El primer programa (figura 20.23) lee dos enteros del teclado, almacena los valores en las variables `a` y `b`, calcula e imprime su suma (almacenada en la variable `c`).

El programa de la figura 20.24 determina e imprime el mayor de dos enteros. Los enteros se introducen desde el teclado y se almacenan en `s` y `t`. La instrucción `if/goto` evalúa la condición `s >= t`. Si es verdadera, el control se transfiere a la línea `90` y se muestra el valor de `s` en pantalla; en caso contrario se muestra `t` y el control se transfiere a la instrucción `end` de la línea `99`, en donde termina el programa.

Simple no cuenta con una instrucción de repetición (como las instrucciones `for`, `while` o `do...while` de Java). Sin embargo, Simple puede simular cada una de las instrucciones de repetición de Java mediante el uso de las instrucciones `if/goto` y `goto`. En la figura 20.25 se utiliza un ciclo controlado por centinela para calcular los cuadrados de varios enteros. Cada entero se introduce desde el teclado y se almacena en la variable `j`. Si el valor introducido es el valor centinela `-9999`, el control se transfiere a la línea `99`, en donde termina el programa. En caso contrario, a `k` se le asigna el cuadrado de `j`, `k` se muestra en pantalla y el control se pasa a la línea `20`, en donde se introduce el siguiente entero.

Utilizando los programas de ejemplo de las figuras 20.23 a 20.25 como guía, escriba un programa de Simple para realizar cada una de las siguientes acciones:

a) Introducir tres enteros, determinar su promedio e imprimir el resultado.
b) Usar un ciclo controlado por centinela para introducir 10 enteros, calcular e imprimir su suma.
c) Usar un ciclo controlado por centinela para introducir 7 enteros, algunos positivos y otros negativos, calcular e imprimir su promedio.

```
10 rem determinar e imprimir la suma de dos enteros
15 rem
20 rem introducir los dos enteros
30 input a
40 input b
45 rem
50 rem sumar los enteros y almacenar el resultado en c
60 let c = a + b
65 rem
70 rem imprimir el resultado
80 print c
90 rem terminar la ejecución del programa
99 end
```

Figura 20.23 Programa de Simple que determina la suma de dos enteros.

```
10 rem determinar e imprimir el mayor de dos enteros
20 input s
30 input t
32 rem
35 rem evaluar si s >= t
40 if s >= t goto 90
45 rem
50 rem t es mayor que s, por lo que se imprime t
60 print t
70 goto 99
75 rem
80 rem s es mayor o igual que t, por lo que se imprime s
90 print s
99 end
```

Figura 20.24 Programa de Simple que encuentra el mayor de dos enteros.

```
10 rem calcular los cuadrados de varios enteros
20 input j
23 rem
25 rem evaluar el valor centinela
30 if j == -9999 goto 99
33 rem
35 rem calcular el cuadrado de j y asignar el resultado a k
40 let k = j * j
50 print k
53 rem
55 rem iterar para obtener el siguiente valor de j
60 goto 20
99 end
```

Figura 20.25 Calcular los cuadrados de varios enteros.

d) Introducir una serie de enteros, determinar e imprimir el mayor. El primer entero introducido indica cuántos números deben procesarse.
e) Introducir 10 enteros e imprimir el menor.
f) Calcular e imprimir la suma de los enteros pares del 2 al 30.
g) Calcular e imprimir el producto de los enteros impares del 1 al 9.

http://librosysolucionarios.net

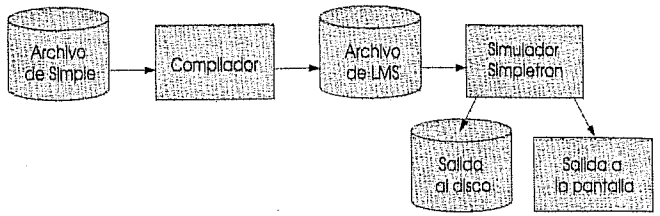


Figura 20.26 Cómo escribir, compilar y ejecutar un programa en lenguaje Simple.

20.27 (*Construcción de un compilador; prerrequisitos: completar los ejercicios 7.43, 7.44, 20.12, 20.13 y 20.26.*) Ahora que hemos presentado el lenguaje Simple (ejercicio 20.26), hablaremos sobre cómo construir un compilador de Simple. Primero debemos considerar el proceso mediante el cual un programa de Simple se convierte a LMS y se ejecuta por el simulador Simpletron (vea la figura 20.26). El compilador lee un archivo que contiene un programa de Simple y lo convierte en código de LMS. Este código se envía a un archivo en disco, en el que las instrucciones de LMS aparecen una en cada línea. Después el archivo de LMS se carga en el simulador Simpletron y los resultados se envían a un archivo en disco y a la pantalla. Observe que el programa de Simpletron desarrollado en el ejercicio 7.44 acepta su entrada mediante el teclado. Este programa debe modificarse para que lea desde un archivo y así pueda ejecutar los programas producidos por nuestro compilador.

El compilador de Simple realiza dos *pasadas* del programa de Simple para convertirlo en LMS. En la primera pasada se construye una *tabla de símbolos* (objeto) en la que cada *número de línea* (objeto), *nombre de variable* (objeto) y *constante* (objeto) del programa de Simple se guarda con su tipo y ubicación correspondiente en el código final de LMS (la tabla de símbolos se describe detalladamente a continuación). En la primera pasada también se produce(n) el (los) objeto(s) correspondientes a la instrucción de LMS para cada una de las instrucciones de Simple (objeto, etcétera). Si el programa de Simple contiene instrucciones que transfieren el control a una línea que se encuentra más adelante en el programa, la primera pasada produce un programa de LMS que contiene algunas instrucciones "no terminadas". En la segunda pasada del compilador se localizan y completan las instrucciones no terminadas, y se envía el programa de LMS a un archivo.

Primera pasada

El compilador empieza leyendo una instrucción del programa de Simple y la coloca en memoria. La línea debe separarse en sus *tokens* individuales (es decir, "piezas" de una instrucción) para su procesamiento y compilación. (Puede usarse la clase `StreamTokenizer` del paquete `java.io`.) Recuerde que todas las instrucciones empiezan con un número de línea, seguido de un comando. A medida que el compilador divide una instrucción en tokens, si éste es un número de línea, una variable o una constante, se coloca en la tabla de símbolos. Un número de línea se coloca en la tabla de símbolos solamente si es el primer token en una instrucción. El objeto `tablaDeSimbolos` es un arreglo de objetos `entradaTabla` que representan a cada uno de los símbolos en el programa. No hay restricción en cuanto al número de símbolos que pueden aparecer en el programa. Por lo tanto, la `tablaDeSimbolos` para un programa específico podría ser extensa. Por ahora, haga que la `tablaDeSimbolos` sea un arreglo de 100 elementos. Usted podrá incrementar o decrementar su tamaño una vez que el programa esté ejecutándose.

Cada objeto `entradaTabla` contiene tres campos. El campo `simbolo` es un entero que contiene la representación Unicode de una variable (recuerde que los nombres de las variables son caracteres individuales), un número de línea o una constante. El campo `tipo` es uno de los siguientes caracteres que indican el tipo de ese símbolo: `'C'` para constante, `'L'` para número de línea o `'V'` para variable. El campo `ubicacion` contiene la ubicación de memoria Simpletron (`00` a `99`) a la que hace referencia el símbolo. La memoria Simpletron es un arreglo de 100 enteros en donde se almacenan instrucciones y datos de LMS. Para un número de línea, la ubicación es el elemento en el arreglo de memoria Simpletron en el que empiezan las instrucciones de LMS para la instrucción de Simple. Para una variable o constante, la ubicación es el elemento en el arreglo de memoria Simpletron en el que se almacena esa variable o constante. Las variables y constantes se asignan desde el final de la memoria Simpletron hacia atrás. La primera variable o constante se almacena en la ubicación `99`, la siguiente en la ubicación `98` y así, sucesivamente.

La tabla de símbolos juega una parte integral para convertir los programas de Simple a LMS. En el capítulo 7 aprendimos que una instrucción de LMS es un entero de cuatro dígitos, compuesto de dos partes: el *código de la operación* y el *operando*. El código de operación se determina mediante los comandos en Simple. Por ejemplo, el comando `input` de Simple corresponde al código de operación `10` de LMS (lectura), y el comando `print` de Simple corresponde al código

de operación 11 de LMS (escritura). El operando es una ubicación en memoria que contiene los datos sobre los cuales el código de operación lleva a cabo su tarea (por ejemplo, el código de operación 10 lee un valor desde el teclado y lo guarda en la ubicación de memoria especificada por el operando). El compilador busca en la `tablaDeSimbolos` para determinar la ubicación de memoria Simpletron para cada símbolo, de manera que pueda utilizarse la ubicación correspondiente para completar las instrucciones de LMS.

La compilación de cada instrucción de Simple se basa en su comando. Por ejemplo, después de insertar el número de línea de una instrucción `rem` en la tabla de símbolos, el compilador ignora el resto de la instrucción ya que un comentario es sólo para fines de documentación. Las instrucciones `input`, `print`, `goto` y `end` corresponden a las instrucciones *leer*, *escribir*, *ramificar* (hacia una ubicación específica) y *parar* de LMS. Las instrucciones que contienen estos comandos de Simple se convierten directamente a LMS. *(Nota:* Una instrucción `goto` puede contener una referencia no resuelta, si el número de línea especificado hace referencia a una instrucción que se encuentre más adelante en el archivo del programa de Simple; a esto se le conoce algunas veces como referencia adelantada.)

Cuando una instrucción `goto` se compila con una referencia no resuelta, la instrucción de LMS debe *marcarse* para indicar que la segunda pasada del compilador debe completar la instrucción. Las banderas se almacenan en un arreglo de 100 elementos llamado `banderas` de tipo `int`, en donde cada elemento se inicializa con `-1`. Si la ubicación de memoria a la que hace referencia un número de línea en el programa de Simple no se conoce todavía (es decir, que no se encuentra en la tabla de símbolos), el número de línea se almacena en el arreglo `banderas`, en el elemento con el mismo índice que la instrucción incompleta. El operando de la instrucción incompleta se establece en `00` temporalmente. Por ejemplo, una instrucción de ramificación incondicional (que hace una referencia adelantada) se deja como `+4000` hasta la segunda pasada del compilador. En breve describiremos la segunda pasada del compilador.

La compilación de las instrucciones `if/goto` y `let` es más complicada que las otras instrucciones; son las únicas instrucciones que producen más de una instrucción de LMS. Para una instrucción `if/goto`, el compilador produce código para evaluar la condición y ramificar hacia otra línea, en caso de ser necesario. El resultado de la ramificación podría ser una referencia no resuelta. Cada uno de los operadores relacionales y de igualdad pueden simularse mediante el uso de las instrucciones *branch zero* y *branch negative* de LMS (o posiblemente una combinación de ambas).

Para una instrucción `let`, el compilador produce código para evaluar una expresión aritmética arbitrariamente compleja que consiste de variables y/o constantes enteras. Las expresiones deben separar cada operando y operador con espacios. En los ejercicios 20.12 y 20.13 se presentaron el algoritmo de conversión de infijo a postfijo y el algoritmo de evaluación de expresiones postfijo que utilizan los compiladores para evaluar expresiones. Antes de proseguir con su compilador, debe completar cada uno de estos ejercicios. Cuando un compilador encuentra una expresión, la convierte de notación infijo a notación postfijo y después evalúa la expresión postfijo.

¿Cómo es que el compilador produce el lenguaje máquina para evaluar una expresión que contiene variables? El algoritmo de evaluación de expresiones postfijo contiene un "gancho" en donde el compilador puede generar instrucciones de LMS, en vez de evaluar la expresión. Para permitir este "gancho" en el compilador, el algoritmo de evaluación postfijo debe modificarse para buscar en la tabla de símbolos cada símbolo que vaya encontrando (y posiblemente insertarlo), determinar la ubicación de memoria correspondiente de ese símbolo y *meter la ubicación de memoria a la pila (en vez del símbolo)*. Cuando se encuentra un operador en la expresión postfijo, las dos ubicaciones de memoria que se encuentran en la parte superior de la pila se sacan y se produce el lenguaje máquina para llevar a cabo la operación, utilizando las ubicaciones de memoria como operandos. El resultado de cada subexpresión se almacena en una ubicación temporal en memoria y se mete de vuelta a la pila, de manera que pueda continuar la evaluación de la expresión postfijo. Al completarse esta evaluación, la ubicación de memoria que contiene el resultado es la única ubicación que queda en la pila. Esta ubicación se saca y se generan las instrucciones de LMS para asignar el resultado a la variable que está a la izquierda de la instrucción `let`.

Segunda pasada

En la segunda pasada del compilador se llevan a cabo dos tareas: Resolver cualquier referencia no resuelta y enviar el código de LMS a un archivo. La resolución de las referencias ocurre así:

a) Buscar en el arreglo `banderas` una referencia no resuelta (es decir, un elemento con un valor distinto de `-1`).
b) Localizar el objeto en el arreglo `tablaDeSimbolos` que contenga el símbolo almacenado en el arreglo `banderas` (asegúrese que el tipo del símbolo sea `'L'` para un número de línea).
c) Insertar la ubicación de memoria del campo `ubicación` en la instrucción con la referencia no resuelta (recuerde que una instrucción que contiene una referencia no resuelta tiene el operando `00`).
d) Repetir los pasos (a), (b) y (c) hasta que se llegue al final del arreglo `banderas`.

Una vez completo el proceso de resolución, todo el arreglo que contiene el código de LMS se envía a un archivo en disco, con una instrucción de LMS por línea. El simulador Simpletron puede leer este archivo para ejecutarlo (una vez que se modifique el simulador para que lea su entrada desde un archivo). El compilar su primer programa de Simple de un archivo de LMS y luego ejecutar ese archivo le dará un verdadero sentido de satisfacción personal.

Un ejemplo completo

En el siguiente ejemplo mostramos la conversión completa de un programa de Simple a LMS, como lo deberá realizar el compilador de Simple. Considere un programa de Simple que recibe un entero y suma los valores desde 1 hasta ese entero. El programa y las instrucciones de LMS producidas por la primera pasada del compilador de Simple se muestran en la figura 20.27 La tabla de símbolos construida por la primera pasada se muestra en la figura 20.28.

La mayoría de las instrucciones de Simple se convierten directamente a instrucciones de LMS individuales. Las excepciones en este programa son los comentarios, la instrucción `if/goto` en la línea `20` y las instrucciones `let`. Los comentarios no se traducen a lenguaje máquina. Sin embargo, el número de línea para un comentario se coloca en la tabla de símbolos, en caso de que se haga referencia al número de línea en una instrucción `goto` o `if/goto`. En la línea `20` del programa se especifica que, si la condición `y == x` es verdadera, el control del programa se transfiere a la línea `60`. Ya que la línea `60` aparece más adelante en el programa, la primera pasada del compilador todavía no ha colocado el valor `60` en la tabla de símbolos. (Se colocan números de línea en la tabla de símbolos solamente cuando aparecen como el primer token en una instrucción.) Por lo tanto, no es posible en este momento determinar el operando de la instrucción *branch zero* de LMS que se encuentra en la ubicación `03` del arreglo de instrucciones de LMS. El compilador coloca `60` en la ubicación `03` del arreglo `banderas` para indicar que la instrucción va a completarse en la segunda pasada.

Debemos llevar el registro de la ubicación de la siguiente instrucción en el arreglo de LMS, ya que no hay una correspondencia de uno a uno entre las instrucciones de Simple y las instrucciones de LMS. Por ejemplo, la instrucción `if/goto` de la línea `20` se compila en tres instrucciones de LMS. Cada vez que se produce una instrucción, debemos incrementar el *contador de instrucciones* a la siguiente ubicación en el arreglo de LMS. Hay que tener en cuenta que el tamaño de la memoria Simpletron podría presentar un problema para los programas de Simple con muchas instrucciones, variables y constantes. Es posible que el compilador se quede sin memoria. Para probar este caso, su programa debe contener un *contador*

Programa de Simple	Ubicación e instrucción de LMS	Descripción
`5 rem sumar 1 a x`	*ninguna*	`rem` se ignora
`10 input x`	`00 +1099`	leer `x` y colocar su valor en la ubicación `99`
`15 rem verificar si y == x`	*ninguna*	`rem` se ignora
`20 if y == x goto 60`	`01 +2098`	cargar `y` (`98`) en el acumulador
	`02 +3199`	restar `x` (`99`) al acumulador
	`03 +4200`	ramificar a ubicación no resuelta si el resultado es cero
`25 rem incrementar y`	*ninguna*	`rem` se ignora
`30 let y = y + 1`	`04 +2098`	cargar `y` en el acumulador
	`05 +3097`	sumar `1` (`97`) al acumulador
	`06 +2196`	almacenar en ubicación temporal `96`
	`07 +2096`	cargar de ubicación temporal `96`
	`08 +2198`	almacenar acumulador en `y`
`35 rem sumar y al total`	*ninguna*	`rem` se ignora
`40 let t = t + y`	`09 +2095`	cargar `t` (`95`) en el acumulador
	`10 +3098`	sumar `y` al acumulador
	`11 +2194`	almacenar en ubicación temporal `94`
	`12 +2094`	cargar de ubicación temporal `94`
	`13 +2195`	almacenar acumulador en `t`
`45 rem ciclo con y`	*ninguna*	`rem` se ignora
`50 goto 20`	`14 +4001`	ramificar a ubicación `01`
`55 rem mostrar resultado`	*ninguna*	`rem` se ignora
`60 print t`	`15 +1195`	mostrar `t` en la pantalla
`99 end`	`16 +4300`	terminar la ejecución

Figura 20.27 Las instrucciones de LMS producidas después de la primera pasada del compilador.

http://librosysolucionarios.net

Símbolo	Tipo	Ubicación
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Figura 20.28 Tabla de símbolos para el programa de la figura 20.27.

de datos para llevar un registro de la ubicación en la que se almacenará la siguiente variable o constante en el arreglo de LMS. Si el valor del contador de instrucciones es mayor que el valor del contador de datos, significa que el arreglo de LMS está lleno. En este caso, el proceso de compilación debe terminar y el compilador debe imprimir un mensaje de error, indicando que se agotó la memoria durante la compilación. Esto sirve para enfatizar que, aunque el programador está libre de la carga que representa para el compilador tener que administrar la memoria, debe determinar cuidadosamente la colocación de instrucciones y datos en la de ella, y debe comprobar que no haya errores como el agotamiento de la memoria durante el proceso de compilación.

El proceso de compilación, paso a paso

Ahora analicemos el proceso de compilación para el programa de Simple que aparece en la figura 20.27. El compilador lee la primera línea del programa:

```
5 rem sumar 1 a x
```

en memoria. El primer token en la instrucción (el número de línea) se determina mediante el uso de la clase `StringTokenizer`. (En el capítulo 11 se describe el uso de esta clase.) El token devuelto por el objeto `StringTokenizer` se convierte en un entero mediante el uso del método `static Integer.parseInt()`, de manera que el símbolo `5` puede localizarse en la tabla de símbolos. Si el símbolo no se encuentra, se inserta en la tabla de símbolos.

Como nos encontramos en el inicio del programa y ésta es la primera línea, todavía no hay símbolos en la tabla. Por lo tanto, se inserta el `5` en la tabla de símbolos con el tipo `L` (número de línea) y se le asigna la primera ubicación en el arreglo de LMS (`00`). Aunque esta línea es un comentario, de todas formas se asigna un espacio en la tabla de símbolos para el número de línea (en caso que se haga referencia al mismo en una instrucción `goto` o `if/goto`). No se genera ninguna instrucción de LMS para una instrucción `rem`, por lo que no se incrementa el contador de instrucciones. Ahora la instrucción:

```
10 input x
```

se divide en tokens. El número de línea `10` se coloca en la tabla de símbolo con el tipo `L` y se le asigna la primera ubicación en el arreglo de LMS (`00` pues, como un comentario empezó el programa, el contador de instrucciones sigue siendo `00`). El comando `input` indica que el siguiente token es una variable (sólo puede aparecer una variable en una instrucción `input`). Como `input` corresponde directamente a un código de operación de LMS, el compilador sólo tiene que determinar la ubicación de `x` en el arreglo de LMS. El símbolo `x` no se encuentra en la tabla de símbolos, por lo que se inserta

en ésta como la representación Unicode de `x`, se le asigna el tipo `V` y la ubicación `99` en el arreglo de LMS (el almacenamiento de datos empieza en la ubicación `99` y se van asignando ubicaciones en forma regresiva). Ahora puede generarse el código LMS para esta instrucción. El código de operación `10` (código de operación de lectura de LMS) se multiplica por `100` y se agrega la ubicación de `x` (según lo determinado en la tabla de símbolos) para completar la instrucción. Después la instrucción se almacena en el arreglo de LMS, en la ubicación `00`. El contador de instrucciones se incrementa en uno, ya que se produjo una sola instrucción de LMS.

Ahora la instrucción:

```
15 rem verificar si y == x
```

se divide en tokens. Se busca en la tabla de símbolos el número de línea `15` (el cual no se encuentra). El número de línea se inserta con el tipo `L` y se le asigna la siguiente ubicación en el arreglo, `01`. (Recuerde que las instrucciones `rem` no producen código, por lo que no se incrementa el contador de instrucciones.)

Ahora la instrucción:

```
20 if y == x goto 60
```

se divide en tokens. El número de línea `20` se inserta en la tabla de símbolos y se le asigna el tipo `L` en la siguiente ubicación en el arreglo de LMS (`01`). El comando `if` indica que se va a evaluar una condición. La variable `y` no se encuentra en la tabla de símbolos, por lo que se inserta, se le asigna el tipo `V` y la ubicación `98`. A continuación se generan las instrucciones de LMS para evaluar la condición. Como no hay un equivalente directo en LMS para la instrucción `if/goto`, ésta debe simularse mediante un cálculo en el que se utilicen `x` y `y`, y después debe hacerse una ramificación con base en el resultado. Si `y` es igual a `x` el resultado de restar `x` a `y` es cero, por lo que puede utilizarse la instrucción *branch zero* con el resultado del cálculo para simular la instrucción `if/goto`. El primer paso requiere que se cargue `y` (de la ubicación `98` del arreglo de LMS) en el acumulador. Esto produce la instrucción `01 +2098`. Luego, se resta `x` del acumulador. Esto produce la instrucción `02 +3199`. El valor en el acumulador puede ser cero, positivo o negativo. Como el operador es `==`, queremos utilizar la operación *branch zero*. Primero se busca en la tabla de símbolos la ubicación de la ramificación (`60` en este caso), la cual no se encuentra. Por lo tanto, `60` se coloca en el arreglo `banderas` en la ubicación `03`, y se genera la instrucción `03 +4200`. (No podemos agregar la ubicación de la ramificación, ya que todavía no hemos asignado una ubicación a la línea `60` en el arreglo de LMS.) El contador de instrucciones se incrementa en `04`.

El compilador continúa con la instrucción:

```
25 rem incrementar y
```

El número de línea `25` se inserta en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `04` en el arreglo de LMS.

Cuando la instrucción:

```
30 let y = y + 1
```

se divide en tokens, el número de línea `30` se inserta en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `04` en el arreglo de LMS. El comando `let` indica que la línea es una instrucción de asignación. Primero se insertan todos los símbolos de la línea en la tabla de símbolos (si no es que están ya ahí). El entero `1` se agrega a la tabla de símbolos con el tipo `C` y se le asigna la ubicación `97`. A continuación, el lado derecho de la asignación se convierte de notación infijo a postfijo. Luego se evalúa la expresión postfijo (`y 1 +`). El símbolo `y` se encuentra ya en la tabla de símbolos y su ubicación correspondiente en memoria se mete a la pila. El símbolo `1` también se encuentra ya en la tabla de símbolos y su ubicación correspondiente en memoria se mete a la pila. Al llegar al operador `+`, el evaluador de expresiones postfijo saca el elemento superior de la pila y lo coloca en el operando derecho del operador, saca de nuevo el elemento superior de la pila, lo coloca en el operando izquierdo del operador y produce las instrucciones de LMS.

```
04 +2098 (load y)
05 +3097 (add 1)
```

El resultado de la expresión se almacena en una ubicación temporal en memoria (`96`) mediante la instrucción:

```
06 +2196 (almacenar temporalmente)
```

y la ubicación temporal se mete en la pila. Ahora que se ha evaluado la expresión, el resultado debe almacenarse en `y` (es decir, la variable del lado izquierdo de `=`). Entonces la ubicación temporal se carga en el acumulador y éste se almacena en `y` mediante las instrucciones:

```
07 +2096 (cargar temporalmente)
08 +2198 (store y)
```

http://librosysolucionarios.net

El lector observará inmediatamente que las instrucciones de LMS parecen ser redundantes. Hablaremos sobre esta cuestión en breve.

Cuando la instrucción:

```
35 rem sumar y al total
```

se divide en tokens, el número de línea `35` se inserta en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `09`.

La instrucción:

```
40 let t = t + y
```

es similar a la línea `30`. La variable `t` se inserta en la tabla de símbolos con el tipo `V` y se le asigna la ubicación `95` en el arreglo de LMS. Las instrucciones siguen la misma lógica y formato que la línea `30`, y se generan las instrucciones `09 +2095`, `10 +3098`, `11 +2194`, `12 +2094` y `13 +2195`. Observe que el resultado de `t + y` se asigna a la ubicación temporal `94` antes de asignarse a `t` (`95`). Una vez más, el lector observará que las instrucciones en las ubicaciones de memoria `11` y `12` parecen ser redundantes. De nuevo, hablaremos sobre esta cuestión en breve.

La instrucción:

```
45 rem ciclo con y
```

es un comentario, por lo que la línea `45` se agrega a la tabla de símbolos con el tipo `L` y se le asigna la ubicación `14` en el arreglo de LMS.

La instrucción:

```
50 goto 20
```

transfiere el control a la línea `20`. El número de línea `50` se inserta en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `14` en el arreglo de LMS. El equivalente de `goto` en LMS es la instrucción de *ramificación incondicional* (`40`), la cual transfiere el control a una ubicación específica en el arreglo de LMS. El compilador busca en la tabla de símbolos la línea `20` y encuentra que a ésta le corresponde la ubicación `01` en el arreglo de LMS. El código de operación (`40`) se multiplica por `100` y se le agrega la ubicación `01` para producir la instrucción `14 +4001`.

La instrucción:

```
55 rem mostrar resultado
```

es un comentario, por lo que la línea `55` se inserta en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `15` en el arreglo de LMS.

La instrucción:

```
60 print t
```

es una instrucción de salida. El número de línea `60` se inserta en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `15` en el arreglo de LMS. El equivalente de `print` en LMS es el código de operación `11` (*write*). La ubicación de `t` se determina a partir de la tabla de símbolos y se agrega al resultado de la multiplicación del código de operación por `100`.

La instrucción:

```
99 end
```

es la línea final del programa. El número de línea `99` se almacena en la tabla de símbolos con el tipo `L` y se le asigna la ubicación `16` en el arreglo de LMS. El comando `end` produce la instrucción de LMS `+4300` (`43` significa *halt* en LMS), la cual se escribe como instrucción final en el arreglo de memoria de LMS.

Esto completa la primera pasada del compilador. Ahora consideremos la segunda pasada. En el arreglo `banderas` se buscan valores distintos de `-1`. La ubicación `03` contiene `60`, por lo que el compilador sabe que la instrucción `03` está incompleta. El compilador completa la instrucción buscando en la tabla de símbolos el número `60`, determinando su ubicación y agregándola a la instrucción incompleta. En este caso la búsqueda determina que la línea `60` corresponde a la ubicación `15` en el arreglo de LMS, por lo que se produce la instrucción completa `03 +4215` que sustituye a `03 +4200`. Ahora el programa de Simple se ha compilado con éxito.

Para crear el compilador, tendrá que llevar a cabo cada una de las siguientes tareas:

a) Modifique el programa simulador de Simpletron que escribió en el ejercicio 7.44 para que reciba la entrada de un archivo especificado por el usuario (vea el capítulo 17). El simulador debe mostrar sus resultados en un archivo en disco, con el mismo formato que el de la pantalla. Convierta el simulador para que sea un programa orientado a objetos. En especial, haga que cada parte del hardware sea un objeto. Ordene los tipos de instrucciones en una jerarquía de clases por medio de la herencia. Después ejecute el programa en forma polimórfica, indicando a cada instrucción que se ejecute a sí misma con un mensaje `ejecutarInstruccion`.

b) Modifique el algoritmo de conversión de expresiones infijo a postfijo del ejercicio 20.12 para procesar operandos enteros con varios dígitos y operandos de nombres de variables con una sola letra. (*Sugerencia:* Puede utilizarse la clase `StringTokenizer` para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros mediante el uso del método `parseInt` de la clase `Integer`.) [*Nota:* La representación de datos de la expresión postfijo debe alterarse para dar soporte a los nombres de variables y constantes enteras.]

c) Modifique el algoritmo de evaluación de expresiones postfijo para procesar operandos enteros con varios dígitos y operandos de nombres de variables. Además, el algoritmo deberá ahora implementar el "gancho" que se describió anteriormente, de manera que se produzcan instrucciones de LMS en vez de evaluar directamente la expresión. (*Sugerencia:* Puede utilizarse la clase `StringTokenizer` para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros mediante el uso del método `parseInt` de la clase `Integer`.) (*Nota:* La representación de datos de la expresión postfijo debe alterarse para dar soporte a los nombres de variables y constantes enteras.)

d) Construya el compilador. Incorpore las partes b) y c) para evaluar las expresiones en instrucciones `let`. Su programa debe contener un método que realice la primera pasada del compilador y un método que realice la segunda pasada del compilador. Ambos métodos pueden llamar a otros métodos para realizar sus tareas. Haga que su compilador esté lo más orientado a objetos que sea posible.

20.28 (*Optimización del compilador de Simple.*) Cuando se compila un programa y se convierte en LMS, se genera un conjunto de instrucciones. Ciertas combinaciones de instrucciones a menudo se repiten, por lo general en tercias conocidas como *producciones*. Una producción normalmente consiste de tres instrucciones tales como *load*, *add* y *store*. Por ejemplo, en la figura 20.29 se muestran cinco de las instrucciones de LMS que se produjeron en la compilación del programa de la figura 20.27. Las primeras tres instrucciones son la producción que suma `1` a `y`. Observe que las instrucciones `06` y `07` almacenan el valor del acumulador en la ubicación temporal `96` y cargan el valor de vuelta en el acumulador, de manera que la instrucción `08` pueda almacenar el valor en la ubicación `98`. A menudo una producción va seguida de una instrucción *load* para la misma ubicación en la que fue guardada. Este código puede *optimizarse* mediante la eliminación de la instrucción *store* y la instrucción *load* que le sigue, las cuales operan en la misma ubicación, con lo que se permite al simulador Simpletron ejecutar el programa con más

```
1 04 +2098 (load)
2 05 +3097 (add)
3 06 +2196 (store)
4 07 +2096 (load)
5 08 +2198 (store)
```

Figura 20.29 Código sin optimizar del programa de la figura 19.25.

Programa de Simple	Ubicación e instrucción de LMS	Descripción
`5 rem sumar 1 a x`	*ninguna*	`rem` se ignora
`10 input x`	`00 +1099`	leer `x` y colocar su valor en la ubicación `99`
`15 rem verificar si y == x`	*ninguna*	`rem` se ignora
`20 if y == x goto 60`	`01 +2098`	cargar `y` (`98`) en el acumulador
	`02 +3199`	restar `x` (`99`) al acumulador
	`03 +4211`	ramificar a ubicación `11` si vale cero
`25 rem incrementar y`	*ninguna*	`rem` se ignora
`30 let y = y + 1`	`04 +2098`	cargar `y` en el acumulador
	`05 +3097`	sumar `1` (`97`) al acumulador
	`06 +2198`	almacenar acumulador en `y` (`98`)
`35 rem sumar y al total`	*ninguna*	`rem` se ignora

Figura 20.30 Código optimizado para el programa de la figura 20.27. (Parte 1 de 2)

http://librosysolucionarios.net

Programa de Simple	Ubicación e instrucción de LMS	Descripción
`40 let t = t + y`	`07 +2096`	cargar `t` de la ubicación (`96`)
	`08 +3098`	sumar `y` (`98`) al acumulador
	`09 +2196`	almacenar acumulador en `t` (`96`)
`45 rem ciclo con y`	*ninguna*	`rem` se ignora
`50 goto 20`	`10 +4001`	ramificar a ubicación `01`
`55 rem mostrar resultado`	*ninguna*	`rem` se ignora
`60 print t`	`11 +1196`	mostrar `t` (`96`) en la pantalla
`99 end`	`12 +4300`	terminar la ejecución

Figura 20.30 Código optimizado para el programa de la figura 20.27. (Parte 2 de 2)

rapidez. En la figura 20.30 se muestra el LMS optimizado para el programa de la figura 20.27. Observe que hay cuatro instrucciones menos en el código optimizado; un ahorro de espacio en memoria del 25%.

20.29 (*Modificaciones al compilador de Simple.*) Realice las siguientes modificaciones al compilador de Simple. Algunas de estas modificaciones podrían requerir también que se modifique el programa simulador de Simpletron que se escribió en el ejercicio 7.44.

a) Permitir el uso del operador residuo (`%`) en instrucciones `let`. El Lenguaje máquina Simpletron debe modificarse para incluir una instrucción residuo.

b) Permitir la exponenciación en una instrucción `let` mediante el uso de `^` como operador de exponenciación. El Lenguaje máquina Simpletron debe modificarse para incluir una instrucción de exponenciación.

c) Permitir al compilador que reconozca letras mayúsculas y minúsculas en instrucciones de Simple (por ejemplo, `'A'` es equivalente a `'a'`). No se requieren modificaciones al simulador de Simpletron.

d) Permitir que las instrucciones `input` lean valores para múltiples variables, tal como `input x, y`. No se requieren modificaciones al simulador Simpletron para llevar a cabo esta mejora en el compilador de Simple.

e) Permitir que el compilador muestre múltiples valores en una sola instrucción `print` como `print a, b, c`. No se requieren modificaciones al simulador de Simpletron para llevar a cabo esta mejora.

f) Agregar al compilador la capacidad de comprobar la sintaxis, de manera que se muestren mensajes de error cuando se encuentren errores de sintaxis en un programa de Simple. No se requieren modificaciones al simulador de Simpletron.

g) Permitir arreglos de enteros. No se requieren modificaciones al simulador de Simpletron para llevar a cabo esta mejora.

h) Permitir subrutinas especificadas por los comandos `gosub` y `return` de Simple. El comando `gosub` pasa el control del programa a una subrutina y el comando `return` pasa el control de vuelta a la instrucción que va después de `gosub`. Esto es similar a la llamada a un método en Java. La misma subrutina puede llamarse desde muchos comandos `gosub` distribuidos a lo largo de un programa. No se requieren modificaciones al simulador de Simpletron.

i) Permitir instrucciones de repetición de la forma:

```
for x = 2 to 10 step 2
instrucciones de Simple
next
```

Esta instrucción `for` realiza iteraciones del `2` al `10` con un incremento de `2`. La línea `next` indica el final del cuerpo de la instrucción `for`. No se requieren modificaciones al simulador de Simpletron.

j) Permitir instrucciones de repetición de la forma:

```
for x = 2 to 10
instrucciones de Simple
next
```

Esta instrucción `for` realiza iteraciones del `2` al `10` con un incremento predeterminado de `1`. No se requieren modificaciones al simulador de Simpletron.

k) Permitir que el compilador procese operaciones de entrada y salida con cadenas. Para ello se requiere la modificación del simulador Simpletron para que procese y almacene valores de cadena. [*Sugerencia:* Cada palabra

de Simpletron (es decir, ubicación de memoria) puede dividirse en dos grupos, cada uno de los cuales almacena un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal Unicode de un carácter. Agregue una instrucción de lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria Simpletron. La primera mitad de la palabra Simpletron en esa ubicación es un conteo del número de caracteres en la cadena (es decir, la longitud de la misma). Cada mitad de palabra subsiguiente contiene un carácter Unicode expresado mediante dos dígitos decimales. La instrucción de lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente.]

l) Permitir que el compilador procese valores de punto flotante, además de enteros. El simulador Simpletron también debe modificarse para procesar valores de punto flotante.

20.30 (*Un intérprete de Simple.*) Un intérprete es un programa que lee la instrucción de un programa escrito en un lenguaje de alto nivel, determina la operación que va a realizar esa instrucción y la ejecuta inmediatamente. El programa en lenguaje de alto nivel no se convierte primero en lenguaje máquina. Los intérpretes se ejecutan con más lentitud que los compiladores, ya que cada instrucción que se encuentra en el programa que va a interpretarse debe primero descifrarse en tiempo de ejecución. Si las instrucciones están contenidas dentro de un ciclo, se descifran cada vez que se encuentran en éste. Las primeras versiones del lenguaje de programación Basic se implementaron como intérpretes. La mayoría de los programas de Java se ejecutan mediante un intérprete.

Escriba un intérprete para el lenguaje Simple descrito en el ejercicio 20.26. El programa debe utilizar el convertidor de expresiones infijo a postfijo que se desarrolló en el ejercicio 20.12, junto con el evaluador de expresiones postfijo que se desarrolló en el ejercicio 20.13, para evaluar las expresiones en una instrucción `let`. Las mismas restricciones impuestas sobre el lenguaje Simple en el ejercicio 20.26 se aplican también a este programa. Pruebe el intérprete con los programas de Simple que se escribieron en el ejercicio 20.26. Compare los resultados de ejecutar estos programas en el intérprete con los resultados de compilar los programas de Simple y ejecutarlos en el simulador Simpletron que se construyó en el ejercicio 7.44.

20.31 (*Insertar/eliminar en cualquier parte de una lista enlazada.*) Nuestra clase de lista enlazada permite inserciones y eliminaciones sólo en la parte frontal y en la parte posterior de la lista enlazada. Estas capacidades eran convenientes para nosotros cuando utilizamos la herencia o la composición para producir una clase pila y una clase cola con una mínima cantidad de código, con sólo reutilizar la clase lista. Normalmente, las listas enlazadas son más generales que las que nosotros vimos. Modifique la clase lista enlazada que desarrollamos en este capítulo para permitir inserciones y eliminaciones en cualquier parte de la lista.

20.32 (*Listas y colas sin referencias a la parte final.*) En nuestra implementación de una lista enlazada (figura 20.3) utilizamos un `primerNodo` y un `ultimoNodo`. El `ultimoNodo` es útil para los métodos `insertarAlFinal` y `eliminarDelFinal` de la clase `Lista`. El método `insertarAlFinal` corresponde al método `enqueue` de la clase `Cola`.

Vuelva a escribir la clase `Lista` de manera que no utilice un `ultimoNodo`. De esta forma, cualquier operación en la parte final de la lista deberá empezar buscando en la lista desde su parte inicial. ¿Afecta esto a nuestra implementación de la clase `Cola` (figura 20.13)?

20.33 (*Rendimiento de los procesos de ordenamiento y búsqueda en árboles binarios.*) Un problema con el ordenamiento de árboles binarios es que el orden en el que se insertan los datos afecta a la forma del árbol; para la misma colección de datos, distintos ordenamientos pueden producir árboles binarios de formas considerablemente distintas. El rendimiento de los algoritmos de ordenamiento y búsqueda en árboles binarios es susceptible a la forma del árbol binario. ¿Qué forma tendría un árbol binario si sus datos se insertaran en orden ascendente?, ¿en orden descendente?, ¿qué forma debería tener el árbol para lograr un máximo rendimiento en el proceso de búsqueda?

20.34 (*Listas indizadas.*) Como se presentan en el texto, la búsqueda en las listas enlazadas debe llevarse a cabo en forma secuencial. Para las listas extensas, esto puede ocasionar un rendimiento pobre. Una técnica común para mejorar el rendimiento del proceso de búsqueda en las listas es crear y mantener un índice de la lista. Un índice es un conjunto de referencias a lugares clave en la lista. Por ejemplo, una aplicación que busca en una lista extensa de nombres podría mejorar el rendimiento al crear un índice con 26 entradas: una para cada letra del alfabeto. Una operación de búsqueda para un apellido que empiece con 'Y' buscaría entonces primero en el índice para determinar en dónde empiezan las entradas con 'Y' y luego "saltaría" hasta ese punto en la lista para buscar linealmente hasta encontrar el nombre deseado. Esto sería mucho más rápido que buscar en la lista enlazada desde el principio. Utilice la clase `Lista` de la figura 20.3 como la base para una clase `ListaIndizada`.

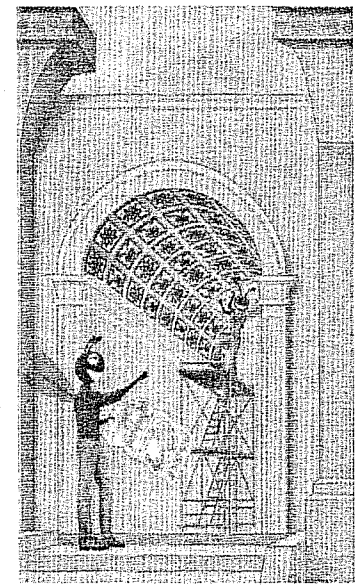
Escriba un programa que demuestre la operación de las listas indizadas. Asegúrese de incluir los métodos `insertarEnListaIndizada`, `buscarEnListaIndizada` y `eliminarDeListaIndizada`.

20.35 En la sección 20.5 creamos una clase de pila a partir de la clase `Lista` mediante la herencia (figura 20.10) y la composición (figura 20.12). En la sección 20.6 creamos una clase de cola a partir de la clase `Lista` mediante la composición (figura 20.13). Cree una clase de cola que herede de la clase `Lista`. ¿Cuáles son las diferencias entre esta clase y la que creamos con la composición?

http://librosysolucionarios.net

21

Paquete de utilerías de Java y manipulación de bits



Objetivos

- Comprender el uso de los contenedores como las clases `Vector` y `Stack`, junto con la interfaz `Enumeration`.
- Utilizar objetos `Hashtable`.
- Utilizar tablas de hash persistentes, manipuladas mediante objetos de la clase `Properties`.
- Utilizar manipulación de bits para procesar los bits individuales en datos de tipo entero.
- Utilizar objetos `BitSet`.

Nada puede tener valor sin ser un objeto de utilidad.
Karl Marx

¡Oh, cuántas flechas
y sólo algunas dieron en el blanco!
Sir Walter Scott

Ahí estaba la puerta para la cual nunca encontré la llave;
ahí estaba el velo a través del cual no podría ver.
Edward Fitzgerald

La reina dijo: "Es un pobre tipo de memoria que sólo funciona al revés".
Lewis Carroll [Charles Lutwidge Dodgson]

La sabiduría se adquiere no con la edad, sino por la capacidad.
Titus Maccius Plautus

http://librosysolucionarios.net

Plan general

21.1 Introducción
21.2 La clase `Vector` y la interfaz `Enumeration`
21.3 La clase `Stack` del paquete `java.util`
21.4 La clase `Hashtable`
21.5 La Clase `Properties`
21.6 Manipulación de bits y operadores a nivel de bits
21.7 La clase `BitSet`

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

21.1 Introducción

En este capítulo hablaremos sobre varias clases e interfaces utilitarias que vienen en el paquete `java.util`, incluyendo la clase `Vector`, la interfaz `Enumeration`, la clase `Stack`, la clase `Hashtable`, la clase `Properties` y la clase `BitSet`.

Los programas utilizan la clase `Vector` para crear objetos tipo arreglos que pueden crecer y reducirse en forma dinámica, a medida que cambian los requerimientos de almacenamiento de datos de un programa. Consideraremos la interfaz `Enumeration`, la cual permite a un programa iterar a través de los elementos de un contenedor tal como un objeto `Vector`. La clase `Stack`, una subclase de `Vector`, ofrece las operaciones convencionales de pila `push` y `pop`, así como algunas otras que no consideramos en el capítulo 20.

La clase `Hashtable` proporciona un marco de trabajo para almacenar datos con claves en tablas, y para recuperar esos datos. En este capítulo explicaremos la teoría de "hashing", una técnica para almacenar y recuperar rápidamente la información de tablas, y demostraremos la clase `Hashtable` de Java. Además consideraremos la clase `Properties`, la cual proporciona el soporte para las tablas de hash persistentes: tablas de hash que pueden escribirse en un archivo mediante un flujo de salida, y que pueden leerse de un archivo mediante un flujo de entrada.

En este capítulo presentaremos también una discusión extensa acerca de los operadores de manipulación de bits, seguida de una discusión acerca de la clase `BitSet`, la cual permite crear objetos tipo arreglos de bits para establecer y obtener valores de bits individuales.

En el capítulo 22, Colecciones, presentaremos un marco de trabajo para manipular grupos de objetos llamados *colecciones*. Los objetos de tipo `Vector`, `Stack` y `Hashtable` son colecciones. Se recomienda que los programas nuevos utilicen las clases e interfaces que se presentan en el capítulo 22. No obstante, muchas de las clases e interfaces que se presentan en este capítulo se utilizan a lo largo de la API de Java y en APIs desarrolladas por terceros. Por esta razón, es bueno estar familiarizado con las clases e interfaces que presentamos aquí. Además, las clases como `Vector` y `Hashtable` se han mejorado con las capacidades que presentamos en el capítulo 22.

21.2 La clase `Vector` y la interfaz `Enumeration`

En la mayoría de los lenguajes de programación, incluyendo Java, los arreglos convencionales son de un tamaño fijo; no pueden crecer ni reducirse en respuesta a los cambios en los requerimientos de almacenamiento de una aplicación. La clase `Vector` proporciona las capacidades de las estructuras de datos tipo arreglos, que pueden cambiar su propio tamaño en forma dinámica.

En cualquier momento, un objeto `Vector` contiene un número de elementos que es menor o igual que su *capacidad*. La capacidad es el espacio que se ha reservado para los elementos de `Vector`. Si un objeto `Vector` requiere de una capacidad adicional, crece en base a un *incremento de capacidad* que usted le especifica, o en base a un incremento de capacidad predeterminado. Si usted no especifica un incremento de capacidad, el sistema duplicará el tamaño de un objeto `Vector` cada vez que se requiera de una capacidad adicional.

http://librosysolucionarios.net

Tip de rendimiento 21.1

Insertar elementos adicionales en un objeto `Vector` cuyo tamaño actual sea menor que su capacidad es una operación relativamente rápida.

Tip de rendimiento 21.2

Es una operación relativamente lenta el insertar un elemento en un objeto `Vector` que necesite crecer más para dar cabida al nuevo elemento.

Tip de rendimiento 21.3

El incremento de capacidad predeterminado duplica el tamaño de un objeto `Vector`. Esto podría parecer un desperdicio de almacenamiento, pero en realidad es una manera eficiente que muchos objetos `Vector` crezcan rápidamente para "tener el tamaño adecuado". Esta operación es mucho más eficiente que hacer que el objeto `Vector` crezca sólo el espacio que necesite cada vez para almacenar un solo elemento. La desventaja es que el objeto `Vector` podría ocupar más espacio del que requiere.

Tip de rendimiento 21.4

Si el espacio de almacenamiento es muy valioso, utilice el método `trimToSize` de `Vector` para reducir la capacidad de un objeto `Vector` a su tamaño exacto. Esta operación optimiza el uso que hace un objeto `Vector` del espacio de almacenamiento. Pero si se agrega otro elemento al objeto `Vector`, éste se verá obligado a crecer en forma dinámica; la operación de reducir el objeto `Vector` no deja espacio para que crezca.

Los objetos `Vector` almacenan referencias a objetos `Object`. Por lo tanto, un programa puede almacenar referencias a cualquier objeto en un objeto `Vector`. Para almacenar valores de tipos primitivos en objetos `Vector`, utilice las clases de tipo de envoltura (por ejemplo, `Integer` y `Double`) del paquete `java.lang` para crear objetos que contengan los valores de tipo primitivo. En la figura 21.1 se demuestra el uso de la clase `Vector` y varios de sus métodos. Para obtener información completa acerca de la clase `Vector`, visite la página Web `java.sun.com/j2se/1.4.1/docs/api/java/util/Vector.html`.

El constructor de la aplicación crea un objeto `Vector` (línea 10) con una capacidad inicial de 10 elementos y una capacidad de incremento de cero (los valores predeterminados para un `Vector`). Este objeto `Vector` duplicará su tamaño cada vez que necesite crecer para alojar más elementos. La clase `Vector` proporciona

```
1  // Fig. 21.1: PruebaVector.java
2  // Uso de la clase Vector.
3  import java.util.*;
4
5  public class PruebaVector {
6     private static final String colores[] = { "rojo", "blanco", "azul" };
7
8     public PruebaVector()
9     {
10       Vector vector = new Vector();
11       imprimirVector( vector ); // imprimir el vector
12
13       // agregar elementos al vector
14       vector.add( "magenta" );
15
16       for ( int cuenta = 0; cuenta < colores.length; cuenta++ )
17          vector.add( colores[ cuenta ] );
18
19       vector.add( "cyan" );
20       imprimirVector( vector ); // imprimir vector
21
22       // mostrar los elementos primero y último
23       try {
```

Figura 21.1 La clase `Vector` del paquete `java.util`. (Parte 1 de 3.)

```
24          System.out.println( "Primer elemento: " + vector.firstElement() );
25          System.out.println( "Ultimo elemento: " + vector.lastElement() );
26       }
27
28       // atrapar excepción si el vector está vacío
29       catch ( NoSuchElementException excepcion ) {
30          excepcion.printStackTrace();
31       }
32
33       // ¿el vector contiene "rojo"?
34       if ( vector.contains( "rojo" ) )
35          System.out.println( "\n\"rojo\" se encontro en el indice " +
36             vector.indexOf( "rojo" ) + "\n" );
37       else
38          System.out.println( "\n\"rojo\" no se encontro\n" );
39
40       vector.remove( "rojo" ); // eliminar la cadena "rojo"
41       System.out.println( "\"rojo\" se ha eliminado" );
42       imprimirVector( vector ); // imprimir el vector
43
44       // ¿el vector contiene "rojo" después de la operación de eliminación?
45       if ( vector.contains( "rojo" ) )
46          System.out.println( "\"rojo\" se encontro en el indice " +
47             vector.indexOf( "rojo" ) );
48       else
49          System.out.println( "\"rojo\" no se encontro" );
50
51       // imprimir el tamaño y la capacidad del vector
52       System.out.println( "\nTamanio: " + vector.size() +
53          "\nCapacidad: " + vector.capacity() );
54
55    } // fin del constructor
56
57    private void imprimirVector( Vector vectorAMostrar )
58    {
59       if ( vectorAMostrar.isEmpty() )
60          System.out.print( "el vector esta vacio" ); // vectorAMostrar está vacío
61
62       else { // iterar a través de los elementos
63          System.out.print( "el vector contiene: " );
64          Enumeration elementos = vectorAMostrar.elements();
65
66          while ( elementos.hasMoreElements() )
67             System.out.print( elementos.nextElement() + " " );
68       }
69
70       System.out.println( "\n" );
71    }
72
73    public static void main( String args[] )
74    {
75       new PruebaVector(); // crear objeto y llamar a su constructor
76    }
77
78 } // fin de la clase PruebaVector
```

Figura 21.1 La clase `Vector` del paquete `java.util`. (Parte 2 de 3.)


```
el vector esta vacio

el vector contiene: magenta rojo blanco azul cyan

Primer elemento: magenta
Ultimo elemento: cyan

"rojo" se encontro en el indice 1

"rojo" se ha eliminado
el vector contiene: magenta blanco azul cyan

"rojo" no se encontro

Tamanio: 4
Capacidad: 10
```

Figura 21.1 La clase `Vector` del paquete `java.util`. (Parte 3 de 3.)

otros tres constructores. El constructor que toma un argumento entero crea un objeto `Vector` vacío con la *capacidad inicial* especificada por ese argumento. El constructor que toma dos argumentos crea un objeto `Vector` con la capacidad inicial especificada por el primer argumento y el *incremento de capacidad* especificado por el segundo argumento. Cada vez que el objeto `Vector` necesite crecer, agregará espacio para el número especificado de elementos en el incremento de capacidad. El constructor que toma un objeto `Collection` crea una copia de los elementos de una colección y los almacena en el objeto `Vector`. En el capítulo 22 veremos los objetos `Collection`.

En las líneas 14, 17 y 19 se hace una llamada al método `add` de `Vector` para agregar objetos (cadenas en este programa) al final del objeto `Vector`. En caso de ser necesario, el objeto `Vector` incrementa su capacidad para alojar al nuevo elemento. La clase `Vector` también cuenta con un método llamado `add` que toma dos argumentos. Este método toma un objeto y un entero, e inserta el objeto en el índice especificado del objeto `Vector`. El método `set` reemplaza el elemento en la posición especificada del objeto `Vector` con un elemento especificado. El método `insertElementAt` crea espacio para el nuevo elemento, desplazando a los elementos existentes.

En la línea 24 se hace una llamada al método `firstElement` de `Vector` para devolver una referencia al primer elemento del objeto `Vector`. En la línea 25 se hace una llamada al método `lastElement` de `Vector` para devolver una referencia al primer elemento del objeto `Vector`. Cada uno de estos métodos lanza una excepción `NoSuchElementException` si no hay elementos en el objeto `Vector` cuando se hacen las llamadas.

En la línea 34 se hace una llamada al método `contains` de `Vector` para determinar si el objeto `Vector` contiene `"rojo"`. Este método devuelve `true` si su argumento se encuentra en el objeto `Vector`; en caso contrario devuelve `false`. El método `contains` utiliza el método `equals` de `Object` para determinar si la clave de búsqueda es igual a uno de los elementos del objeto `Vector`. Muchas clases sobrescriben al método `equals` para llevar a cabo las comparaciones en una manera que sea específica a esas clases. Por ejemplo, la clase `String` declara a `equals` para comparar los caracteres individuales en los dos objetos `String` que se están comparando. Si el método `equals` no se sobrescribe se utiliza la versión original de este método, heredada de la clase `Object`. Esta versión realiza comparaciones que utilizan el operador `==` para determinar si dos referencias se refieren al mismo objeto en memoria.

En la línea 36 se hace una llamada al método `indexOf` de `Vector` para determinar el índice de la primera ubicación en el objeto `Vector` que contiene ese argumento. El método devuelve `-1` si el argumento no se encuentra en el objeto `Vector`. Una versión sobrecargada de este método toma un segundo argumento que especifica el índice en el objeto `Vector` en el que debe empezar la búsqueda.

http://librosysolucionarios.net

Tip de rendimiento 21.5



Los métodos `contains` *e* `indexOf` *de* `Vector` *realizan búsquedas lineales en el contenido de un objeto* `Vector`*; estas estructuras son ineficientes para objetos* `Vector` *grandes. Si un programa busca frecuentemente elementos en una colección, considere el uso de un objeto* `Hashtable` *(sección 21.4) o una de las implementaciones de* `Map` *que vienen en la API de colecciones de Java (capítulo 22).*

En la línea 40 se hace una llamada al método `remove` de `Vector` para eliminar del objeto `Vector` la primera ocurrencia de su argumento. Este método devuelve `true` si encuentra el elemento en el objeto `Vector`; en caso contrario, el método devuelve `false`. Si se elimina el elemento, todos los elementos que se encuentran después de ese elemento en el objeto `Vector` se desplazan una posición hacia el inicio del objeto `Vector` para ocupar la posición del elemento eliminado. La clase `Vector` también proporciona el método `removeAllElements` para eliminar todos los elementos de un objeto `Vector`, y el método `removeElementAt` para eliminar el elemento en un índice especificado.

En las líneas 52 y 53 se utilizan los métodos `size` y `capacity` de `Vector` para determinar el número de elementos que tiene el objeto `Vector` en ese momento y el número de elementos que pueden almacenarse en el objeto `Vector` sin necesidad de asignar más memoria, respectivamente.

En la línea 59 se hace una llamada al método `isEmpty` de `Vector` para determinar si el objeto `Vector` está vacío. Este método devuelve `true` si no hay elementos en el objeto `Vector`; en caso contrario, el método devuelve `false`.

En la línea 64 se hace una llamada al método `elements` de `Vector` para devolver un objeto `Enumeration` que permite al programa iterar a través de los elementos del objeto `Vector`. Un objeto `Enumeration` proporciona dos métodos: `hasMoreElements` y `nextElement`.[1] En la línea 66, el método `hasMoreElements` devuelve `true` si hay más elementos en el objeto `Vector`. En la línea 67, el método `nextElement` devuelve una referencia al siguiente objeto `Object` en el objeto `Vector`. Si no hay más elementos, el método `nextElement` lanza una excepción `NoSuchElementException`.

21.3 La clase `Stack` del paquete `java.util`

En el capítulo 20, Estructuras de datos, aprendimos a crear estructuras de datos fundamentales como listas enlazadas, pilas, colas y árboles. En un mundo de reutilización de software, en vez de crear estructuras de datos a medida que las vamos necesitando, a menudo aprovechamos las ya existentes. En esta sección investigaremos la clase `Stack` incluida en el paquete de utilerías de Java (`java.util`).

En la sección 21.2 hablamos sobre la clase `Vector`, que implementa a un arreglo que puede cambiar su tamaño en forma dinámica. La clase `Stack` extiende a la clase `Vector` para implementar una estructura de datos tipo pila que almacena referencias a objetos `Object`. Para almacenar tipos primitivos, utilice las clases de tipo de envoltura apropiadas del paquete `java.util` para crear un objeto que contenga el valor de tipo primitivo. En la figura 21.2 se demuestra el uso de varios métodos de la clase `Stack`. Para obtener información completa acerca de la clase `Stack`, visite la página Web `java.sun.com/j2se/1.4.1/docs/api/java/util/Stack.html`.

```
1  // Fig. 21.2: PruebaStack.java
2  // Programa para probar la clase java.util.Stack.
3  import java.util.*;
4
5  public class PruebaStack {
6
7     public PruebaStack()
8     {
```

Figura 21.2 La clase `Stack` del paquete `java.util`. (Parte 1 de 3.)

1. En el capítulo 22 presentaremos una interfaz similar llamada `Iterator`, la cual es más flexible que `Enumeration` (también [illegible] mínimente es la opción preferida. Aquí demostraremos los objetos `Enumeration`, ya que se utilizan con algunas de las clases más antiguas en la API de Java.

http://librosysolucionarios.net


```
9        Stack pila = new Stack();
10
11       // crear objetos para almacenarlos en la pila
12       Boolean bool = Boolean.TRUE;
13       Character caracter = new Character( '$' );
14       Integer entero = new Integer( 34567 );
15       String cadena = "hola";
16
17       // usar el método push
18       pila.push( bool );
19       imprimirPila( pila );
20       pila.push( caracter );
21       imprimirPila( pila );
22       pila.push( entero );
23       imprimirPila( pila );
24       pila.push( cadena );
25       imprimirPila( pila );
26
27       // eliminar elementos de la pila
28       try {
29          Object objetoEliminado = null;
30
31          while ( true ) {
32             objetoEliminado = pila.pop(); // usar el método pop
33             System.out.println( objetoEliminado.toString() + " sacado" );
34             imprimirPila( pila );
35          }
36       }
37
38       // atrapar excepción si la pila está vacía al sacar el elemento
39       catch ( EmptyStackException excepcionPilaVacia ) {
40          excepcionPilaVacia.printStackTrace();
41       }
42    }
43
44    private void imprimirPila( Stack pila )
45    {
46       if ( pila.isEmpty() )
47          System.out.print( "la pila esta vacia" ); // la pila está vacía
48
49       else {
50          System.out.print( "la pila contiene: " );
51          Enumeration elementos = pila.elements();
52
53          // iterar a través de los elementos
54          while ( elementos.hasMoreElements() )
55             System.out.print( elementos.nextElement() + " " );
56       }
57
58       System.out.println( "\n" ); // avanzar a la siguiente línea
59    }
60
61    public static void main( String args[] )
62    {
63       new PruebaStack();
```

Figura 21.2 La clase `Stack` del paquete `java.util`. (Parte 2 de 3.)

```
64      }
65
66  } // fin de la clase PruebaStack
```

```
la pila contiene: true

la pila contiene: true $

la pila contiene: true $ 34567

la pila contiene: true $ 34567 hola

hola sacado
la pila contiene: true $ 34567

34567 sacado
la pila contiene: true $

$ sacado
la pila contiene: true

true sacado
la pila está vacía

java.util.EmptyStackException
        at java.util.Stack.peek(Stack.java:79)
        at java.util.Stack.pop(Stack.java:61)
        at PruebaStack.<init>(PruebaStack.java:32)
        at PruebaStack.main(PruebaStack.java:63)
```

Figura 21.2 La clase `Stack` del paquete `java.util`. (Parte 3 de 3.)

En la línea 9 del constructor se crea un objeto `Stack` vacío. En cada una de las líneas 18, 20, 22 y 24 se hace una llamada al método ***`push`*** de `Stack` para agregar objetos a la parte superior de la pila.

En la línea 32 del ciclo `while` infinito se hace una llamada al método ***`pop`*** de `Stack` para eliminar el elemento superior de la pila. El método devuelve una referencia `Object` al elemento eliminado. Si no hay elementos en el objeto `Stack`, el método `pop` lanza una excepción ***`EmptyStackException`***, la cual termina el ciclo en este programa. La clase `Stack` también declara el método ***`peek`***. Este método devuelve el elemento superior de la pila sin sacarlo.

En la línea 46 se hace una llamada al método ***`isEmpty`*** de `Stack` (heredado por `Stack` de la clase `Vector`) para determinar si la pila está vacía. Si está vacía, el método devuelve `true`; en caso contrario, devuelve `false`.

El método `imprimirPila` (líneas 44 a 59) utiliza un objeto `Enumeration` para iterar a través de los elementos en la pila. La parte superior actual de la pila (el último valor que se haya metido a la pila) es el último valor que se imprimirá en pantalla. Como la clase `Stack` extiende a la clase `Vector`, la interfaz `public` completa de la clase `Vector` está disponible para los clientes de la clase `Stack`. Por ejemplo, el método `imprimirPila` invoca al método `elements` (línea 51) para obtener un objeto `Enumeration` de la pila.

Tip para prevenir errores 21.1

Un programa puede llevar a cabo operaciones de objetos `Vector` en objetos `Stack` que no se permitan ordinariamente en estructuras de datos convencionales. Esto podría "corromper" los elementos del objeto. Libro para truir su integridad. Al manipular un objeto `Stack`, sólo deben utilizarse los métodos `push` y `pop` para agregar y eliminar elementos en el objeto `Stack`.

http://librosysolucionarios.net

21.4 La clase `Hashtable`

Los lenguajes de programación orientados a objetos facilitan la creación de nuevos tipos. Cuando un programa crea objetos de tipos nuevos o existentes, necesita administrar esos objetos eficientemente. Esta tarea incluye el almacenamiento y la recuperación de objetos. Los procesos de almacenar y recuperar información con arreglos son eficientes si cierto aspecto de sus datos concuerda directamente con un valor de clave numérico y esas claves son únicas y están estrechamente empaquetadas. Si tiene 100 empleados con números de Seguro Social de nueve dígitos y desea almacenar y recuperar datos utilizando el número de Seguro Social como una clave, necesita utilizar un arreglo con mil millones de elementos, ya que hay mil millones de números únicos de nueve dígitos (000,000,000-999,999,999). Esto no es nada práctico para casi todas las aplicaciones que utilizan números de Seguro Social como claves. Si el programa pudiera tener un arreglo de ese tamaño, podría tener un alto rendimiento para guardar y recuperar registros de empleados si utilizara el número de Seguro Social como índice para el arreglo.

Hay numerosas aplicaciones que tienen este problema, ya sea que las claves sean del tipo incorrecto (por ejemplo, enteros no positivos) o que sean del tipo correcto, pero estén ampliamente distribuidas a lo largo de un rango bastante extenso. Lo que se necesita es un esquema de alta velocidad para convertir claves como los números de Seguro Social, los números de piezas de inventario y demás números similares en índices de arreglo únicos. De esta forma, cuando una aplicación necesite almacenar algo, el esquema podría convertir rápidamente la clave de la aplicación en un índice, y el registro de información podría almacenarse en una posición en el arreglo. El proceso de recuperación de los datos se lleva a cabo de la misma forma: Una vez que la aplicación tiene una clave con la que desea recuperar un registro de datos, la aplicación simplemente aplica la conversión a la clave; esto produce el índice del arreglo en donde están almacenados los datos, con el cual se recuperan.

El esquema que describimos aquí es la base de una técnica llamada *hashing*, o *transformación de claves*. ¿Por qué ese nombre? Cuando convertimos una clave en un índice de arreglo, literalmente desordenamos los bits, formando un tipo de número mezclado o revuelto. En realidad, el número no tiene un significado verdadero más allá de su utilidad para almacenar y recuperar un registro de datos específico.

Se presenta un fallo en el esquema cuando hay *colisiones*; es decir, cuando dos claves distintas se transforman y se colocan en la misma celda (o elemento) en el arreglo. Como no podemos almacenar dos valores en el mismo espacio, necesitamos encontrar un hogar alternativo para todos los valores, aparte del primero, que se transformen en un índice de arreglo específico. Hay muchos esquemas para hacer esto. Uno de ellos es "revolver de nuevo" (es decir, aplicar otra transformación tipo "hashing" a la clave, para proporcionar otra posible celda en el arreglo). El proceso de "hashing" está diseñado para distribuir los valores a lo largo de la tabla, por lo que se supone que, con sólo unas cuantas transformaciones, se encontrará una celda disponible.

En otro esquema se aplica una transformación (hash) para localizar la primera celda disponible. Si esa celda está ocupada, se buscan más celdas en forma lineal hasta encontrar una disponible. El proceso de recuperación de datos se lleva a cabo de la misma forma: La clave se transforma una vez mediante la técnica de "hashing" para determinar la ubicación inicial y verificar si contiene los datos deseados. Si es así, la búsqueda termina. De lo contrario, se busca en las siguientes celdas en forma lineal hasta encontrar los datos disponibles.

La solución más popular a las colisiones en las tablas de hash es hacer que cada celda de la tabla sea una "cubeta" de hash, que generalmente sería una lista enlazada de todos los pares clave-valor que se transforman y dan como resultado esa celda. Ésta es la solución que implementa la clase `Hashtable` de Java (del paquete `java.util`). Un objeto ***`Hashtable`*** asigna *claves* a los *valores*.

El *factor de carga* de una tabla de hash afecta al rendimiento de los esquemas de "hashing". El factor de carga es la proporción entre el número de celdas ocupadas en la tabla de hash y el tamaño de la tabla de hash. Entre más cerca esté la proporción de 1.0, mayor será la probabilidad de que haya colisiones.

Tip de rendimiento 21.6

El factor de carga en una tabla de hash es un ejemplo clásico de la concesión entre espacio en memoria/tiempo de ejecución; Al incrementar el factor de carga obtenemos una mejor utilización de la memoria, pero el programa se ejecutará más lentitud debido al aumento en colisiones de "hashing". Al reducir el factor de carga obtenemos una mejor velocidad en el programa debido a la reducción en colisiones de "hashing", pero empeora la utilización de la memoria debido a que una porción más grande de la tabla de hash permanece vacía.

La complejidad al programar las tablas de hash apropiadamente es alta para la mayoría de los programadores casuales. Los estudiantes de ciencias computacionales estudian esquemas de "hashing" detenidamente, en cursos llamados "Estructuras de datos" y "Algoritmos". Como un reconocimiento al valor que tiene la técnica de "hashing" para la mayoría de los programadores, los diseñadores de Java proporcionaron la clase **Hashtable** para permitir a los programadores utilizar el "hashing" sin necesidad de implementar los complicados detalles.[2]

En la figura 21.3 se proporciona una GUI que utiliza varios métodos de **Hashtable**. En la línea 21 se crea un objeto **Hashtable** vacío, con una capacidad predeterminada (11 elementos) y un factor de carga predeterminado (0.75). Cuando el número de posiciones ocupadas en el objeto **Hashtable** se vuelve mayor que la capacidad multiplicada por el factor de carga, la tabla aumenta de tamaño. La clase **Hashtable** también proporciona un constructor, el cual toma un argumento que especifica la capacidad, y un constructor que toma dos argumentos, los cuales especifican la capacidad y el factor de carga, respectivamente.

```
1  // Fig. 21.3: CuentaTipoPalabras.java
2  // Conteo del número de ocurrencias de cada palabra en una cadena.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7
8  public class CuentaTipoPalabras extends JFrame {
9     private JTextArea campoEntrada;
10    private JLabel indicador;
11    private JTextArea pantalla;
12    private JButton botonIniciar;
13
14    private Hashtable tabla;
15
16    public CuentaTipoPalabras()
17    {
18       super( "Cuenta de tipos de palabras" );
19       campoEntrada = new JTextArea( 3, 20 );
20
21       tabla = new Hashtable();
22
23       botonIniciar = new JButton( "Iniciar" );
24       botonIniciar.addActionListener(
25
26          new ActionListener() { // clase interna anónima
27
28             public void actionPerformed( ActionEvent evento )
29             {
30                crearTabla();
31                pantalla.setText( crearSalida() );
32             }
33
34          } // fin de la clase interna anónima
35
```

Figura 21.3 La clase **Hashtable** del paquete **java.util**. (Parte 1 de 3.)

2. Este concepto es enormemente importante para nuestro estudio de la programación orientada a objetos. Como se dijo en capítulos anteriores, las clases encapsulan y ocultan la complejidad (es decir, los detalles de implementación) y crean componentes amigables para los usuarios. El proceso de fabricar clases apropiadamente para que exhiban dicho comportamiento es una de las habilidades más valiosas en el campo de la programación orientada a objetos.


```
36       ); // fin de la llamada a addActionListener
37
38       indicador = new JLabel( "Escriba una cadena:" );
39       pantalla = new JTextArea( 15, 20 );
40       pantalla.setEditable( false );
41
42       JScrollPane mostrarPanelDesplazable = new JScrollPane( pantalla );
43
44       // agregar componentes a la GUI
45       Container contenedor = getContentPane();
46       contenedor.setLayout( new FlowLayout() );
47       contenedor.add( indicador );
48       contenedor.add( campoEntrada );
49       contenedor.add( botonIniciar );
50       contenedor.add( mostrarPanelDesplazable );
51
52       setSize( 400, 400 );
53       setVisible( true );
54
55    } // fin del constructor
56
57    // crear tabla a partir de la entrada del usuario
58    private void crearTabla() {
59       String entrada = campoEntrada.getText();
60       StringTokenizer palabras = new StringTokenizer( entrada, " \n\t\r" );
61
62        while ( palabras.hasMoreTokens() ) {
63          String palabra = palabras.nextToken().toLowerCase(); // obtener palabra
64
65          // si la tabla contiene la palabra
66          if ( tabla.containsKey( palabra ) ) {
67
68             Integer cuenta = (Integer) tabla.get( palabra ); // obtener el valor
69
70             // e incrementarlo
71             tabla.put( palabra, new Integer( cuenta.intValue() + 1 ) );
72          }
73          else // en caso contrario, agregar la palabra con un valor de 1
74             tabla.put( palabra, new Integer( 1 ) );
75
76       } // fin de instrucción while
77
78    } // fin del método crearTabla
79
80    // crear cadena con valores de tabla
81    private String crearSalida() {
82       String salida = "";
83       Enumeration claves = tabla.keys();
84
85       // iterar a través de las claves
86       while ( claves.hasMoreElements() ) {
87          Object claveActual = claves.nextElement();
88
89          // mostrar los pares clave-valor
90          salida += claveActual + "\t" + tabla.get( claveActual ) + "\n";
```

Figura 21.3 La clase **Hashtable** del paquete **java.util**. (Parte 2 de 3.)

http://librosysolucionarios.net

```
91        }
92
93        salida += "tamaño: " + tabla.size() + "\n";
94        salida += "estaVacia: " + tabla.isEmpty() + "\n";
95
96        return salida;
97
98     } // fin del método crearSalida
99
100    public static void main( String args[] )
101     {
102       CuentaTipoPalabras aplicacion = new CuentaTipoPalabras();
103       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
104    }
105
106 } // fin de la clase CuentaTipoPalabras
```

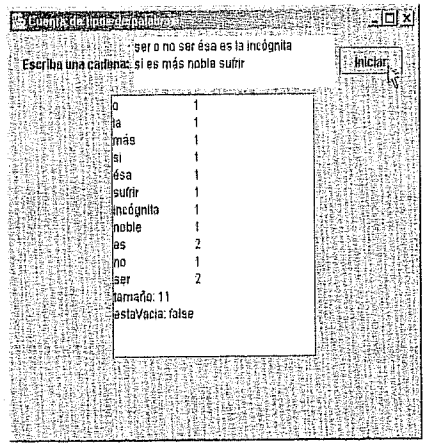


Figura 21.3 La clase **Hashtable** del paquete **java.util**. (Parte 3 de 3.)

Cuando el usuario introduce una cadena y hace clic en **Iniciar**, en la línea 30 se hace una llamada al método **crearTabla** (líneas 58 a 78). En la línea 66 se hace una llamada al método ***containsKey*** de **Hashtable** para determinar si la clave especificada como argumento se encuentra en la tabla de hash (es decir, si hay un valor asociado con esa clave).[3] De ser así, el método devuelve **true**; en caso contrario, el método devuelve **false**. Si la clave no se encuentra en la tabla de hash, en la línea 74 se hace una llamada al método **put** de **Hashtable** para agregar una *clave* (la palabra) y un *valor* (un objeto **Integer** que contiene 1) en la tabla de hash. El método **put** devuelve el valor original para esa clave en la tabla de hash, o **null** si no se ha insertado una clave anteriormente en la tabla de hash; este procedimiento ayuda al programa a manejar los casos en los que se trata de sustituir el valor almacenado para una clave dada. Si la clave o el valor son **null**, se produce una excepción **NullPointerException**.

Si la clave se encuentra en la tabla de hash, en la línea 68 se hace una llamada al método **get** de **Hashtable** para localizar el valor asociado con la clave que se especificó como argumento. Si la clave no está presente en la tabla, el método devolverá **null**. En la línea 71 se hace una llamada al método **put** de **Hashtable** para sustituir la cuenta para la palabra en la tabla de hash.

3. La clase **Hashtable** también proporciona el método ***contains*** para determinar si el objeto **Object** especificado como su argumento se encuentra en el objeto **Hashtable**.

http://librosysolucionarios.net

El método **crearSalida** (líneas 81 a 98) utiliza un objeto **Enumeration** para mostrar todos los valores en la tabla. En la línea 93 se hace una llamada al método **size** para devolver el número de pares clave-valor en la tabla de hash. En la línea 94 se hace una llamada al método ***isEmpty*** para determinar si la tabla de hash está vacía. Para obtener información completa acerca de la clase **Hashtable** y sus métodos, visite la página Web **java.sun.com/j2se/1.4.1/docs/api/java/util/Hashtable.html**.

21.5 La clase Properties

Un objeto ***Properties*** es un objeto **Hashtable** persistente, que normalmente almacena pares clave-valor de cadenas (suponiendo que usted utilice los métodos ***setProperty*** y ***getProperty*** para manipular la tabla, en vez de los métodos **put** y **get** heredados de **Hashtable**. Al decir "persistente", nos referimos a que el objeto **Properties** puede escribirse en un flujo de salida (posiblemente un archivo) y leerse devuelta a través de un flujo de entrada. De hecho, la mayoría de los objetos en Java pueden enviarse y recibirse mediante la serialización de objetos en Java, presentada en el capítulo 17.

Un uso común de los objetos **Properties** en versiones anteriores de Java era para mantener los datos de configuración de las aplicaciones o las preferencias de los usuarios para las aplicaciones. Una nueva característica de la Plataforma Java 2 Standard Edition, versión 1.4, es la *API Preferences*, que está diseñada para reemplazar el uso de la clase **Properties** con un mecanismo más robusto para mantener los datos de configuración y preferencia. Para obtener más información acerca de la API Preferences, visite la página Web **java.sun.com/j2se/1.4.1/docs/guide/lang/preferences.html**.

La clase **Properties** extiende a la clase **Hashtable**, por lo que los objetos **Properties** tienen los métodos descritos en la figura 21.3. La clase **Properties** cuenta con métodos adicionales, los cuales se demuestran en la figura 21.4.

```
1  // Fig. 21.4: PruebaProperties.java
2  // Demuestra el uso de la clase Properties del paquete java.util.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.util.*;
7  import javax.swing.*;
8
9  public class PruebaProperties extends JFrame {
10    private JLabel etiquetaEstado;
11    private Properties tabla;
12    private JTextArea areaPantalla;
13    private JTextField campoValor, campoNombre;
14
15    // configurar la GUI para probar la tabla Properties
16    public PruebaProperties()
17    {
18       super( "Prueba de Properties" );
19
20       tabla = new Properties(); // crear tabla Properties
21
22       Container contenedor = getContentPane();
23
24       // establecer región NORTH del esquema BorderLayout de la ventana
25       JPanel subPanelNorte = new JPanel();
26
27       subPanelNorte.add( new JLabel( "Valor de propiedad" ) );
28       campoValor = new JTextField( 10 );
```

Figura 21.4 La clase **Properties** del paquete **java.util**. (Parte 1 de 6.)

```
29          subPanelNorte.add( campoValor );
30
31          subPanelNorte.add( new JLabel( "Nombre de propiedad (clave)" ) );
32          campoNombre = new JTextField( 10 );
33          subPanelNorte.add( campoNombre );
34
35          JPanel northPanel = new JPanel();
36          northPanel.setLayout( new BorderLayout() );
37          northPanel.add( subPanelNorte, BorderLayout.NORTH );
38
39          etiquetaEstado = new JLabel();
40          northPanel.add( etiquetaEstado, BorderLayout.SOUTH );
41
42          contenedor.add( northPanel, BorderLayout.NORTH );
43
44          // establecer región CENTER del esquema BorderLayout de la ventana
45          areaPantalla = new JTextArea( 4, 35 );
46          contenedor.add( new JScrollPane( areaPantalla ),
47             BorderLayout.CENTER );
48
49          // establecer región SOUTH del esquema BorderLayout de la ventana
50          JPanel southPanel = new JPanel();
51          southPanel.setLayout( new GridLayout( 1, 5 ) );
52
53          // botón para colocar un nuevo par nombre-valor en la tabla Properties
54          JButton botonColocar = new JButton( "Colocar" );
55          southPanel.add( botonColocar );
56
57          botonColocar.addActionListener(
58
59             new ActionListener() { // clase interna anónima
60
61                // colocar par nombre-valor en la tabla Properties
62                public void actionPerformed( ActionEvent evento )
63                {
64                   Object valor = tabla.setProperty(
65                      campoNombre.getText(), campoValor.getText() );
66
67                   if ( valor == null )
68                      showstatus( "Colocar: " + campoNombre.getText() +
69                         " " + campoValor.getText() );
70
71                   else
72                      showstatus( "Colocar: " + campoNombre.getText() + " " +
73                         campoValor.getText() + "; Sustituye a: " + valor );
74
75                   listarPropiedades();
76                }
77
78             } // fin de la clase interna anónima
79
80          ); // fin de la llamada a addActionListener
81
82          // botón para vaciar el contenido de la tabla Properties
83          JButton botonBorrar = new JButton( "Borrar" );
```

Figura 21.4 La clase **Properties** del paquete **java.util**. (Parte 2 de 6.)


```
84          southPanel.add( botonBorrar );
85
86          botonBorrar.addActionListener(
87
88             new ActionListener() { // clase interna anónima
89
90                // usar el método clear para vaciar la tabla
91                public void actionPerformed( ActionEvent evento )
92                {
93                   tabla.clear();
94                   showstatus( "Se borró la tabla en memoria" );
95                   listarPropiedades();
96                }
97
98             } // fin de la clase interna anónima
99
100         ); // fin de la llamada a addActionListener
101
102         // botón para obtener el valor de una propiedad
103         JButton botonObtenerPropiedad = new JButton( "Obtener propiedad" );
104         southPanel.add( botonObtenerPropiedad );
105
106         botonObtenerPropiedad.addActionListener(
107
108            new ActionListener() { // clase interna anónima
109
110               // usar método getProperty para obtener el valor de una propiedad
111               public void actionPerformed( ActionEvent evento )
112               {
113                  Object valor = tabla.getProperty(
114                     campoNombre.getText() );
115
116                  if ( valor != null )
117                     showstatus( "Obtener propiedad: " + campoNombre.getText() +
118                        " " + valor.toString() );
119
120                  else
121                     showstatus( "Obtener: " + campoNombre.getText() +
122                        " no se encuentra en la tabla" );
123
124                  listarPropiedades();
125               }
126
127            } // fin de la clase interna anónima
128
129         ); // fin de la llamada a addActionListener
130
131         // botón para guardar el contenido de la tabla Properties en un archivo
132         JButton botonGuardar = new JButton( "Guardar" );
133         southPanel.add( botonGuardar );
134
135         botonGuardar.addActionListener(
136
137            new ActionListener() { // clase interna anónima
138
```

Figura 21.4 La clase **Properties** del paquete **java.util**. (Parte 3 de 6.)

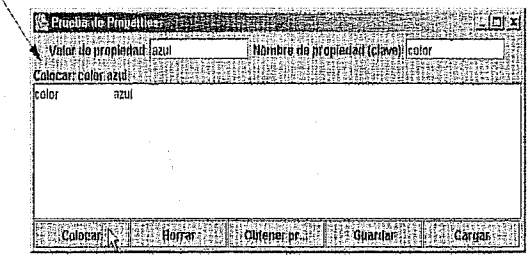
http://librosysolucionarios.net

```
139             // usar el método save para colocar el contenido en un archivo
140             public void actionPerformed( ActionEvent evento )
141             {
142                // guardar el contenido de la tabla
143                try {
144                   FileOutputStream salida =
145                      new FileOutputStream( "props.dat" );
146
147                   tabla.store( salida, "Propiedades de ejemplo" );
148                   salida.close();
149
150                   listarPropiedades();
151                }
152
153                // procesar problemas con la salida al archivo
154                catch( IOException excepcionES ) {
155                   excepcionES.printStackTrace();
156                }
157             }
158
159          } // fin de la clase interna anónima
160
161       ); // fin de la llamada a addActionListener
162
163       // botón para cargar el contenido de la tabla Properties desde un archivo
164       JButton botonCargar = new JButton( "Cargar" );
165       southPanel.add( botonCargar );
166
167       botonCargar.addActionListener(
168
169          new ActionListener() { // clase interna anónima
170
171             // usar el método load para leer el contenido del archivo
172             public void actionPerformed( ActionEvent evento )
173             {
174                // cargar el contenido de la tabla
175                try {
176                   FileInputStream entrada =
177                      new FileInputStream( "props.dat" );
178
179                   tabla.load( entrada );
180                   entrada.close();
181                   listarPropiedades();
182                }
183
184                // procesar problemas con la entrada del archivo
185                catch( IOException excepcionES ) {
186                   excepcionES.printStackTrace();
187                }
188             }
189
190          } // fin de la clase interna anónima
191
192       ); // fin de la llamada a addActionListener
193
```

Figura 21.4 La clase **Properties** del paquete **java.util**. (Parte 4 de 6.)


```
194          contenedor.add( southPanel, BorderLayout.SOUTH );
195
196          setSize( 550, 225 );
197          setVisible( true );
198
199       } // fin del constructor
200
201       // mostrar valores de las propiedades
202       public void listarPropiedades()
203       {
204          StringBuffer bufer = new StringBuffer();
205          String name, valor;
206
207          Enumeration enumeracion = tabla.propertyNames();
208
209          while ( enumeracion.hasMoreElements() ) {
210             name = enumeracion.nextElement().toString();
211             valor = tabla.getProperty( name );
212
213             bufer.append( name ).append( '\t' );
214             bufer.append( valor ).append( '\n' );
215          }
216
217          areaPantalla.setText( bufer.toString() );
218       }
219
220       // mostrar objeto String en la etiqueta etiquetaEstado
221       public void showstatus( String s )
222       {
223          etiquetaEstado.setText( s );
224       }
225
226       public static void main( String args[] )
227       {
228          PruebaProperties aplicacion = new PruebaProperties();
229          aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
230       }
231
232    } // fin de la clase PruebaProperties
```

La etiqueta especifica la operación que se acaba de realizar



http://librosysolucionarios.net

Figura 21.4 La clase **Properties** del paquete **java.util**. (Parte 5 de 6.)

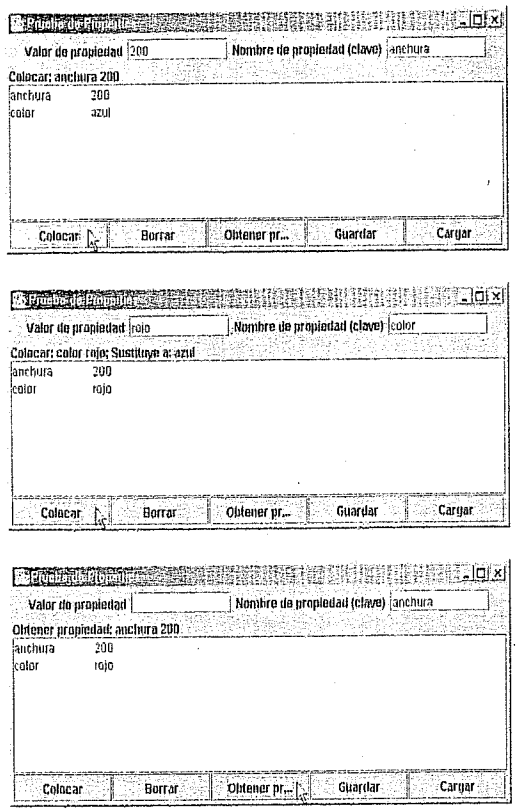


Figura 21.4 La clase `Properties` del paquete `java.util`. (Parte 6 de 6.)

En la línea 20 se utiliza el constructor sin argumentos para crear una tabla `Properties` vacía, sin propiedades predeterminadas. La clase `Properties` también proporciona un constructor sobrecargado que recibe una referencia a un objeto `Properties`, el cual contiene valores predeterminados para las propiedades.

En las líneas 64 y 65 se hace una llamada al método `setProperty` de `Properties`, para almacenar un valor para la clave especificada. Si la clave no existe en la tabla, `setProperty` devuelve `null`; en caso contrario, devuelve el valor anterior para esa clave.

En las líneas 113 y 114 se hace una llamada al método `getProperty` de `Properties`, para localizar el valor asociado con la clave especificada. Si la clave no se encuentra en este objeto `Properties`, `getProperty` trata de localizar la clave en el objeto `Properties` predeterminado (si hay uno). El proceso continúa recursivamente hasta que no haya más objetos `Properties` predeterminados (recuerde que todo objeto `Properties` puede inicializarse con un objeto `Properties` predeterminado); en este momento, `getProperty` devuelve `null`. Una versión sobrecargada de este método recibe un segundo argumento que especifica el valor predeterminado a devolver si `getProperty` no puede localizar la clave.

En la línea 147 se hace una llamada al método *`store`* de `Properties` para guardar el contenido del objeto `Properties`, enviándolo al objeto `OutputStream` especificado como el primer argumento (en este caso, un objeto `FileOutputStream`). El argumento `String` es una descripción del objeto `Properties`. La clase `Properties` también proporciona el método *`list`*, el cual toma un argumento `PrintStream`. Este método es útil para mostrar el conjunto de propiedades.

http://libros

Tip para prevenir errores 21.2

Use el método `list` de `Properties` para mostrar el contenido de un objeto `Properties`, para fines de depuración.

En la línea 179 se hace una llamada al método *`load`* de `Properties` para restaurar el contenido del objeto `Properties`, del objeto `InputStream` especificado como el primer argumento (en este caso, un objeto `FileInputStream`). En la línea 207 se hace una llamada al método *`propertyNames`* de `Properties` para obtener un objeto `Enumeration`, el cual contiene una enumeración de los nombres de las propiedades. El valor de cada propiedad puede determinarse mediante el uso del método `getProperty`.

21.6 Manipulación de bits y los operadores a nivel de bits

Java proporciona muchas herramientas de manipulación de bits para los programadores que necesitan llegar hasta el nivel de "bits y bytes". Los sistemas operativos, el software de equipos de prueba, el software de red y muchos otros tipos de software requieren que el programador se comunique "directamente con el hardware". Ahora hablaremos sobre las herramientas de manipulación de bits de Java y los operadores a nivel de bits.

Las computadoras representan todos los datos internamente como secuencias de bits. Cada bit puede asumir el valor `0` o el valor `1`. En la mayoría de los sistemas, una secuencia de ocho bits forma un byte: la unidad de almacenamiento estándar para una variable de tipo `byte`. Otros tipos se almacenan en números de bytes más grandes. Los operadores a nivel de bits pueden manipular los bits de operandos integrales (es decir, operaciones de tipo `byte`, `char`, `short`, `int` y `long`), pero no operandos de punto flotante.

Observe que las discusiones de los operadores a nivel de bits en esta sección muestran las representaciones binarias de los operandos enteros. Para una explicación detallada del sistema numérico binario (también conocido como base 2), vea el apéndice C, Sistemas numéricos.

Los operadores a nivel de bits son *AND a nivel de bits* (`&`), *OR incluyente a nivel de bits* (`|`), *OR excluyente a nivel de bits* (`^`), *desplazamiento a la izquierda* (`<<`), *desplazamiento a la derecha con signo* (`>>`), *desplazamiento a la derecha sin signo* (`>>>`) y *complemento a nivel de bits* (`~`). Los operadores AND a nivel de bits, OR incluyente a nivel de bits y OR excluyente a nivel de bits comparan sus operandos, bit por bit. El operador AND a nivel de bits hace que cada bit en el resultado sea `1` si, y sólo si el bit correspondiente en ambos operandos es `1`. El operador OR incluyente a nivel de bits hace que cada bit en el resultado sea `1` si el bit correspondiente en un operando (o ambos) es `1`. El operador OR excluyente a nivel de bits hace que cada bit en el resultado sea `1` si el bit correspondiente en sólo uno de los operandos es `1`. El operador de desplazamiento a la izquierda desplaza los bits de su operando izquierdo a la izquierda, según el número de bits especificados en su operando derecho. El operador de desplazamiento a la derecha con signo desplaza los bits en su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho; si el operando izquierdo es negativo, los `1`s se desplazan desde la izquierda; en caso contrario, los `0`s se desplazan desde la izquierda. El operador de desplazamiento a la derecha sin signo desplaza los bits de su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho; los `0`s se desplazan desde la izquierda. El operador de complemento a nivel de bits hace que todos los bits que sean `0` en su operando sean `1` en el resultado, y hace que todos los bits que sean `1` en su operando sean `0` en el resultado. Los operadores a nivel de bits se sintetizan en la figura 21.5.

Operador	Nombre	Descripción
`&`	AND a nivel de bits	Los bits en el resultado se hacen `1` si los bits correspondientes en ambos operandos son `1`.
`|`	OR incluyente a nivel de bits	Los bits en el resultado se hacen `1` si cuando menos uno de los bits correspondientes en los dos operandos es `1`.
`^`	OR excluyente a nivel de bits	Los bits en el resultado se hacen `1` si cuando menos uno de los bits correspondientes en los dos operandos es `1`.

Figura 21.5 Operadores a nivel de bits. (Parte 1 de 2.)

www.librosysolucionarios.net

Operador	Nombre	Descripción
<<	desplazamiento a la izquierda	Desplaza los bits del primer operando a la izquierda, según el número de bits especificados por el segundo operando; rellenar desde la derecha con bits en 0.
>>	desplazamiento a la derecha con signo	Desplaza los bits del primer operando a la derecha, según el número de bits especificados por el segundo operando. Si el primer operando es negativo, se rellenan los espacios con 1s desde la izquierda; en caso contrario, se rellenan los espacios con 0s desde la izquierda.
>>>	desplazamiento a la derecha sin signo	Desplaza los bits del primer operando a la derecha, según el número de bits especificados por el segundo operando; los espacios se rellenan con 0s desde la izquierda.
~	complemento a nivel de bits	Todos los bits que sean 0 se hacen 1, y todos los bits que sean 1 se hacen 0.

Figura 21.5 Operadores a nivel de bits. (Parte 2 de 2.)

Al utilizar los operadores a nivel de bits, es conveniente mostrar los valores en su representación binaria para ilustrar los efectos de esos operadores. La aplicación de la figura 21.6 permite al usuario introducir un entero en un objeto **JTextField**. El método **actionPerformed** (líneas 29 a 33) lee la cadena del objeto **JTextField**, la convierte en un entero e invoca al método **obtenerBits** (líneas 52 a 73) para obtener una representación de cadena del entero, en bits. El resultado se muestra en el objeto **campoSalida**. El entero se muestra en su representación binaria, en grupos de ocho bits cada uno. En la línea 63 del método **obtenerBits** se utiliza el operador AND a nivel de bits para combinar las variables **valor** y **mascaraMostrar**. A menudo se utiliza el operador AND con un operando conocido como *máscara* (un valor entero con ciertos bits en 1). Las máscaras se utilizan para ocultar ciertos bits y seleccionar otros en un valor. En **obtenerBits**, la variable máscara **mascaraMostrar** recibe el valor **1 << 31**, o:

```
10000000 00000000 00000000 00000000
```

El operador de desplazamiento a la izquierda desplaza el valor 1 desde el bit menos significativo (más a la derecha) hasta el bit más significativo (más a la izquierda) en **mascaraMostrar**, y rellena con 0s empezando de la derecha.

```
1  // Fig. 21.6: ImprimirBits.java
2  // Imprimir un entero sin signo en bits.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class ImprimirBits extends JFrame {
8     private JTextField campoSalida;
9
10    // configurar la GUI
11    public ImprimirBits()
12    {
13       super( "Imprimir representaciones de números en bits" );
14
15       Container contenedor = getContentPane();
16       contenedor.setLayout( new FlowLayout() );
17
```

Figura 21.6 Imprimir los bits en un entero. (Parte 1 de 3.)


```
18       contenedor.add( new JLabel( "Escriba un entero " ) );
19
20       // campo de texto para leer un valor del usuario
21       JTextField campoEntrada = new JTextField( 10 );
22       contenedor.add( campoEntrada );
23
24       campoEntrada.addActionListener(
25
26          new ActionListener() { // clase interna anónima
27
28             // leer entero y obtener representación a nivel de bits
29             public void actionPerformed( ActionEvent evento )
30             {
31                int valor = Integer.parseInt( evento.getActionCommand() );
32                campoSalida.setText( obtenerBits( valor ) );
33             }
34
35          } // fin de la clase interna anónima
36
37       ); // fin de la llamada a addActionListener
38
39       contenedor.add( new JLabel( "El entero en bits es" ) );
40
41       // campo de texto para mostrar entero en formato a nivel de bits
42       campoSalida = new JTextField( 33 );
43       campoSalida.setEditable( false );
44       contenedor.add( campoSalida );
45
46       setSize( 720, 70 );
47       setVisible( true );
48
49    } // fin del constructor
50
51    // mostrar representación en bits del valor int especificado
52    private String obtenerBits( int valor )
53    {
54       // crear valor int con 1 en el bit más a la izquierda y 0s en las demás posiciones
55       int mascaraMostrar = 1 << 31;
56
57       StringBuffer bufer = new StringBuffer( 35 ); // bufer de salida
58
59       // para cada bit, anexar 0 o 1 al búfer
60       for ( int bit = 1; bit <= 32; bit++ ) {
61
62          // usar mascaraMostrar para aislar el bit
63          bufer.append( ( valor & mascaraMostrar ) == 0 ? '0' : '1' );
64
65          valor <<= 1; // desplazar valor una posición a la izquierda
66
67          if ( bit % 8 == 0 )
68             bufer.append( ' ' ); // anexar un espacio al búfer cada 8 bits
69       }
70
71       return bufer.toString();
72
```

http://librosysolucionarios.net

Figura 21.6 Imprimir los bits en un entero. (Parte 2 de 3.)

```
73    } // fin del método obtenerBits
74
75    public static void main( String args[] )
76    {
77       ImprimirBits aplicacion = new ImprimirBits();
78       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
79    }
80
81 } // fin de la clase ImprimirBits
```

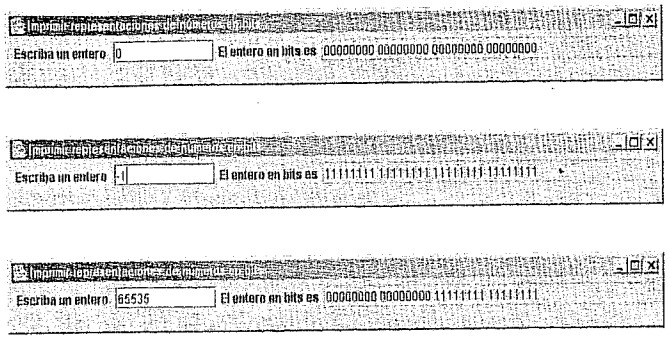


Figura 21.6 Imprimir los bits en un entero. (Parte 3 de 3.)

En la línea 63 se determina si el bit más a la izquierda de la variable `valor` en ese momento es un `1` o un `0`, y anexa `'1'` o `'0'`, respectivamente, a `bufer`. Suponga que `valor` contiene `2000000000` (`01110111 00110101 10010100 00000000`). Cuando `valor` y `mascaraMostrar` se combinan mediante el uso de `&`, todos los bits excepto el bit más significativo (más a la izquierda) en la variable `valor` se "enmascaran" (ocultan), ya que cualquier bit al que se le aplique la operación AND con un `0` se convierte en `0`. Si el bit más a la izquierda es `1`, la expresión `valor & mascaraMostrar` se evalúa dando como resultado un número distinto de cero, y en la línea 63 se anexa `'1'`; en caso contrario, en la línea 63 se anexa `'0'`. Después, en la línea 65 se desplaza un bit a la izquierda el valor de la variable `valor`, mediante la expresión `valor <<= 1`. (Esta expresión es equivalente a `valor = valor << 1`.) Estos pasos se repiten para cada uno de los bits en la variable `valor`. Al final del método `obtenerBits`, en la línea 71 se convierte el objeto `StringBuffer` en `String` y se devuelve este último. (*Nota*: La clase `Integer` proporciona el método `toBinaryString`, el cual devuelve una cadena que contiene la representación binaria de un entero.) En la figura 21.7 se sintetizan los resultados producidos al combinar dos bits con el operador AND a nivel de bits (`&`).



Error común de programación 21.1

Utilizar el operador AND condicional (`&&`) en vez del operador AND a nivel de bits (`&`).

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Figura 21.7 Combinación de dos bits mediante el operador AND a nivel de bits (`&`).

http://librosysolucionarios.net

En la figura 21.8 se demuestra el uso de los operadores AND a nivel de bits, OR incluyente a nivel de bits, OR excluyente a nivel de bits y complemento a nivel de bits. El programa utiliza el método `obtenerBits` (líneas 160 a 181) para obtener una representación de cadena de los valores enteros. El programa permite a los usuarios introducir valores en objetos `JTextField` (para los operadores binarios, deben introducirse dos valores) y oprimir el botón que representa la operación que desean probar. El programa muestra el resultado de cada operación en representación entera y a nivel de bits.

```
1  // Fig. 21.8: OpsBitsMisc.java
2  // Uso de los operadores a nivel de bits.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class OpsBitsMisc extends JFrame {
8     private JTextField campoEntrada1, campoEntrada2,
9        campoBits1, campoBits2, campoBits3, campoResultado;
10    private int valor1, valor2;
11
12    // configurar la GUI
13    public OpsBitsMisc()
14    {
15       super( "Operadores a nivel de bits" );
16
17       JPanel panelEntrada = new JPanel();
18       panelEntrada.setLayout( new GridLayout( 4, 2 ) );
19
20       panelEntrada.add( new JLabel( "Escriba 2 enteros" ) );
21       panelEntrada.add( new JLabel( "" ) );
22
23       panelEntrada.add( new JLabel( "Valor 1" ) );
24       campoEntrada1 = new JTextField( 8 );
25       panelEntrada.add( campoEntrada1 );
26
27       panelEntrada.add( new JLabel( "Valor 2" ) );
28       campoEntrada2 = new JTextField( 8 );
29       panelEntrada.add( campoEntrada2 );
30
31       panelEntrada.add( new JLabel( "Resultado" ) );
32       campoResultado = new JTextField( 8 );
33       campoResultado.setEditable( false );
34       panelEntrada.add( campoResultado );
35
36       JPanel panelBits = new JPanel();
37       panelBits.setLayout( new GridLayout( 4, 1 ) );
38       panelBits.add( new JLabel( "Representaciones de bits" ) );
39
40       campoBits1 = new JTextField( 33 );
41       campoBits1.setEditable( false );
42       panelBits.add( campoBits1 );
43
44       campoBits2 = new JTextField( 33 );
45       campoBits2.setEditable( false );
46       panelBits.add( campoBits2 );
```

Figura 21.8 Los operadores AND a nivel de bits, OR incluyente a nivel de bits, OR excluyente a nivel de bits y complemento a nivel de bits. (Parte 1 de 5.)

```
47
48          campoBits3 = new JTextField( 33 );
49          campoBits3.setEditable( false );
50          panelBits.add( campoBits3 );
51
52          JPanel panelBotones = new JPanel();
53
54          // botón para realizar operación con AND a nivel de bits
55          JButton botonAND = new JButton( "AND" );
56          panelBotones.add( botonAND );
57
58          botonAND.addActionListener(
59
60             new ActionListener() { // clase interna anónima
61
62                // realizar operación AND a nivel de bits y mostrar resultados
63                public void actionPerformed( ActionEvent evento )
64                {
65                   establecerCampos();
66                   campoResultado.setText( Integer.toString( valor1 & valor2 ) );
67                   campoBits3.setText( obtenerBits( valor1 & valor2 ) );
68                }
69
70             } // fin de la clase interna anónima
71
72          ); // fin de la llamada a addActionListener
73
74          // botón para realizar operación OR incluyente a nivel de bits
75          JButton botonORIncluyente = new JButton( "OR incluyente" );
76          panelBotones.add( botonORIncluyente );
77
78          botonORIncluyente.addActionListener(
79
80             new ActionListener() { // clase interna anónima
81
82                // realizar operación OR incluyente a nivel de bits y mostrar resultados
83                public void actionPerformed( ActionEvent evento )
84                {
85                   establecerCampos();
86                   campoResultado.setText( Integer.toString( valor1 | valor2 ) );
87                   campoBits3.setText( obtenerBits( valor1 | valor2 ) );
88                }
89
90             } // fin de la clase interna anónima
91
92          ); // fin de la llamada a addActionListener
93
94          // botón para realizar operación OR excluyente a nivel de bits
95          JButton botonORExcluyente = new JButton( "OR excluyente" );
96          panelBotones.add( botonORExcluyente );
97
98          botonORExcluyente.addActionListener(
99
100            new ActionListener() { // clase interna anónima
```

http://librosysoluciones.net

Figura 21.8 Los operadores AND a nivel de bits, OR incluyente a nivel de bits, OR excluyente a nivel de bits y complemento a nivel de bits. (Parte 2 de 5.)


```
101
102               // realizar operación OR excluyente a nivel de bits y mostrar resultados
103               public void actionPerformed( ActionEvent evento )
104               {
105                  establecerCampos();
106                  campoResultado.setText( Integer.toString( valor1 ^ valor2 ) );
107                  campoBits3.setText( obtenerBits( valor1 ^ valor2 ) );
108               }
109
110            } // fin de la clase interna anónima
111
112         ); // fin de la llamada a addActionListener
113
114         // botón para realizar operación complemento a nivel de bits
115         JButton botonComplemento = new JButton( "Complemento" );
116         panelBotones.add( botonComplemento );
117
118         botonComplemento.addActionListener(
119
120            new ActionListener() { // clase interna anónima
121
122               // realizar operación complemento a nivel de bits y mostrar resultados
123               public void actionPerformed( ActionEvent evento )
124               {
125                  campoEntrada2.setText( "" );
126                  campoBits2.setText( "" );
127
128                  int valor = Integer.parseInt( campoEntrada1.getText() );
129
130                  campoResultado.setText( Integer.toString( ~valor ) );
131                  campoBits1.setText( obtenerBits( valor ) );
132                  campoBits3.setText( obtenerBits( ~valor ) );
133               }
134
135            } // fin de la clase interna anónima
136
137         ); // fin de la llamada a addActionListener
138
139         Container contenedor = getContentPane();
140         contenedor.add( panelEntrada, BorderLayout.WEST );
141         contenedor.add( panelBits, BorderLayout.EAST );
142         contenedor.add( panelBotones, BorderLayout.SOUTH );
143
144         setSize( 600, 150 );
145         setVisible( true );
146
147      } // fin del constructor
148
149      // mostrar números y su forma en bits
150      private void establecerCampos()
151      {
152         valor1 = Integer.parseInt( campoEntrada1.getText() );
153         valor2 = Integer.parseInt( campoEntrada2.getText() );
154
```

Figura 21.8 Los operadores AND a nivel de bits, OR incluyente a nivel de bits, OR excluyente a nivel de bits y complemento a nivel de bits. (Parte 3 de 5.)

```
155         campoBits1.setText( obtenerBits( valor1 ) );
156         campoBits2.setText( obtenerBits( valor2 ) );
157      }
158
159      // mostrar representación de bits del valor int especificado
160      private String obtenerBits( int valor )
161      {
162         // crear valor int con 1 en el bit más a la izquierda y 0s en las demás posiciones
163         int mascaraMostrar = 1 << 31;
164
165         StringBuffer bufer = new StringBuffer( 35 ); // búfer para la salida
166
167         // para cada bit anexar 0 o 1 al búfer
168         for ( int bit = 1; bit <= 32; bit++ ) {
169
170            // usar mascaraMostrar para aislar bit
171            bufer.append( ( valor & mascaraMostrar ) == 0 ? '0' : '1' );
172
173            valor <<= 1; // desplazar valor una posición a la izquierda
174
175            if ( bit % 8 == 0 )
176               bufer.append( ' ' ); // anexar un espacio al búfer cada 8 bits
177         }
178
179         return bufer.toString();
180
181      } // fin del método obtenerBits
182
183      public static void main( String args[] )
184      {
185         OpsBitsMisc aplicacion = new OpsBitsMisc();
186         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
187      }
188
189   } // fin de la clase OpsBitsMisc
```

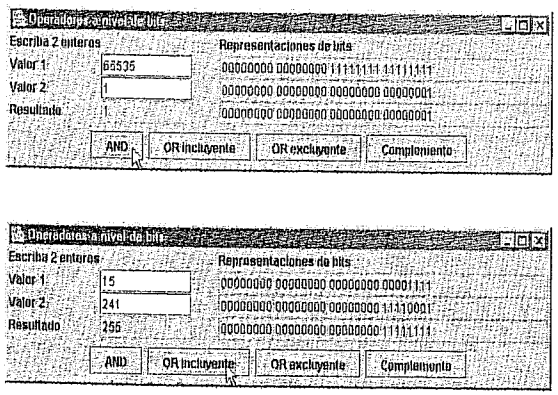


Figura 21.8 Los operadores AND a nivel de bits, OR incluyente a nivel de bits, OR excluyente a nivel de bits y complemento a nivel de bits. (Parte 4 de 5.)

http://librosysolucionarios.net

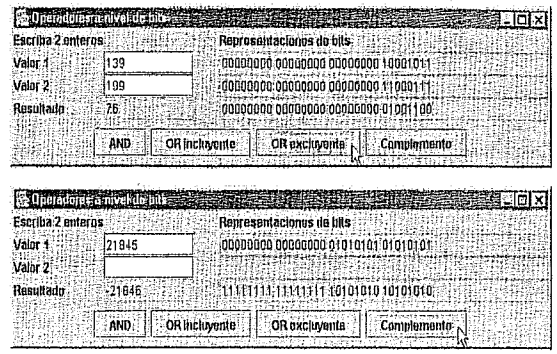


Figura 21.8 Los operadores AND a nivel de bits, OR incluyente a nivel de bits, OR excluyente a nivel de bits y complemento a nivel de bits. (Parte 5 de 5.)

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Figura 21.9 Combinación de dos bits mediante el operador OR incluyente a nivel de bits (|).

La primera ventana de salida en la figura 21.8 muestra el resultado de combinar los valores `65535` y `1` con el operador AND a nivel de bits (`&`; líneas 66 y 67). Todos los bits, excepto el bit menos significativo en el valor `65535`, se "enmascaran" (ocultan) al aplicar la operación AND con el valor `1`.

El operador OR incluyente a nivel de bits (|) hace que ciertos bits sean `1` en un operando. La segunda ventana de salida en la figura 21.8 muestra el resultado de combinar los valores `15` y `241` mediante el uso del operador OR a nivel de bits (líneas 86 y 87); el resultado es `255`. En la figura 21.9 se sintetizan los resultados producidos al combinar dos bits con el operador OR incluyente a nivel de bits.

El operador OR excluyente a nivel de bits (^) hace que cada bit en el resultado sea `1` si *exactamente* uno de los bits correspondientes en sus dos operandos es `1`. La tercera ventana de salida en la figura 21.8 muestra el resultado de combinar los valores `139` y `199` mediante el uso del operador OR excluyente (líneas 106 y 107); el resultado es `76`. En la figura 21.10 se sintetizan los resultados producidos al combinar dos bits con el operador OR excluyente a nivel de bits.

El operador de complemento *a nivel de bits* (~) hace que todos los bits que sean `1` en su operando se conviertan a `0` en el resultado, y que todos los bits que sean `0` en su operando se conviertan a `1` en el resultado; es-

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Figura 21.10 Combinación de dos bits mediante el operador OR excluyente a nivel de bits (^).

to se conoce también como "tomar el *complemento a uno* del valor". La cuarta ventana de salida en la figura 21.8 muestra el resultado de tomar el complemento a uno del valor `21845` (líneas 130 y 132). El resultado es `-21846`.

El programa de la figura 21.11 demuestra el uso de los operadores *desplazamiento a la izquierda* (`<<`), *desplazamiento a la derecha con signo* (`>>`) y *desplazamiento a la derecha sin signo* (`>>>`). El método `obtenerBits` (líneas 114 a 135) obtiene un objeto `String`, el cual contiene la representación de bits de los valores enteros. El programa permite al usuario introducir un entero en un objeto `JTextField` y oprimir *Intro* para mostrar la representación de bits del entero en un segundo objeto `JTextField`. El usuario puede oprimir un botón que representa una operación de desplazamiento, para llevar a cabo un desplazamiento de un bit y ver los resultados de este desplazamiento en representación entera y a nivel de bits.

El operador de desplazamiento a la izquierda (`<<`) desplaza los bits de su operando izquierdo a la izquierda, según el número de bits especificados en su operando derecho (esta operación se lleva a cabo en la línea 56 de la figura 21.11). Los bits de la derecha que quedan vacíos se reemplazan con `0`s; los `1`s que se desplazan más allá de la última posición a la izquierda se pierden. Las primeras cuatro ventanas de salida en la figura 21.11 demuestran el uso del operador de desplazamiento a la izquierda. Empezando con el valor 1, el botón de desplazamiento a la izquierda se oprimió dos veces, produciendo los valores 2 y 4, respectivamente. La cuarta ventana de salida muestra el resultado del valor 1 desplazado 31 veces. Observe que el resultado es un valor negativo. Eso se debe a que un `1` en el bit más significativo se utiliza para indicar un entero con valor negativo.

El operador de desplazamiento a la derecha con signo (`>>`) desplaza los bits de su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho (esta operación se lleva a cabo en la línea 77 de la figura 21.11). Al llevar a cabo un desplazamiento a la derecha, los bits de la izquierda que quedan vacíos se reemplazan por `0`s si el número es positivo, o por `1`s si el número es negativo. Cualquier `1` que se desplace más allá de la última posición de la derecha se pierde. La quinta y sexta ventanas de salida en la figura 21.11 muestran el resultado de desplazar dos veces a la derecha (con extensión de signo) el valor que se muestra en la cuarta ventana de salida.

El operador de desplazamiento a la derecha sin signo (`>>>`) desplaza los bits de su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho (esta operación se lleva a cabo en la línea 98 de la figura 21.11). Al llevar a cabo un desplazamiento a la derecha sin signo, los bits de la izquierda que quedan vacíos se reemplazan por `0`s. Cualquier `1` que se desplace más allá de la última posición de la derecha se pierde. La octava y novena ventanas de salida en la figura 21.11 muestran el resultado de desplazar dos veces a la derecha sin signo el valor que se muestra en la séptima ventana de salida. Cada uno de los operadores a nivel de bits (exceptuando al operador de complemento a nivel de bits) tiene su correspondiente operador de asignación. Estos *operadores de asignación a nivel de bits* se muestran en la figura 21.12.

```
1  // Fig. 21.11: DesplazamientoBits.java
2  // Uso de los operadores de desplazamiento a nivel de bits.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DesplazamientoBits extends JFrame {
8     private JTextField campoBits, campoValor;
9
10    // configurar la GUI
11    public DesplazamientoBits()
12    {
13       super( "Desplazamiento de bits" );
14
15       Container contenedor = getContentPane();
16       contenedor.setLayout( new FlowLayout() );
17
```

Figura 21.11 Operadores de desplazamiento a nivel de bits. (Parte 1 de 5.)


```
18       contenedor.add( new JLabel( "Entero a desplazar " ) );
19
20       // campo de texto para que el usuario introduzca un entero
21       campoValor = new JTextField( 12 );
22       contenedor.add( campoValor );
23
24       campoValor.addActionListener(
25
26          new ActionListener() { // clase interna anónima
27
28             // leer el valor y mostrar su representación a nivel de bits
29             public void actionPerformed( ActionEvent evento )
30             {
31                int valor = Integer.parseInt( campoValor.getText() );
32                campoBits.setText( obtenerBits( valor ) );
33             }
34
35          } // fin de la clase interna anónima
36
37       ); // fin de la llamada a addActionListener
38
39       // campo de texto para mostrar representación a nivel de bits de un entero
40       campoBits = new JTextField( 33 );
41       campoBits.setEditable( false );
42       contenedor.add( campoBits );
43
44       // botón para desplazar bits una posición a la izquierda
45       JButton botonIzquierda = new JButton( "<<" );
46       contenedor.add( botonIzquierda );
47
48       botonIzquierda.addActionListener(
49
50          new ActionListener() { // clase interna anónima
51
52             // desplazar una posición a la izquierda y mostrar nuevo valor
53             public void actionPerformed( ActionEvent evento )
54             {
55                int valor = Integer.parseInt( campoValor.getText() );
56                valor <<= 1;
57                campoValor.setText( Integer.toString( valor ) );
58                campoBits.setText( obtenerBits( valor ) );
59             }
60
61          } // fin de la clase interna anónima
62
63       ); // fin de la llamada a addActionListener
64
65       // botón para desplazar el valor una posición a la derecha con signo
66       JButton botonDerechaConSigno = new JButton( ">>" );
67       contenedor.add( botonDerechaConSigno );
68
69       botonDerechaConSigno.addActionListener(
70
71          new ActionListener() { // clase interna anónima
72
```

http://librosysolucionarios.net

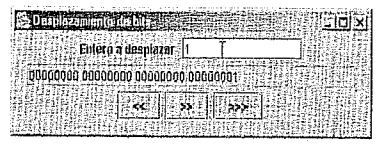
Figura 21.11 Operadores de desplazamiento a nivel de bits. (Parte 2 de 5.)

```
73             // desplazar una posición a la derecha y mostrar el nuevo valor
74             public void actionPerformed( ActionEvent evento )
75             {
76                int valor = Integer.parseInt( campoValor.getText() );
77                valor >>= 1;
78                campoValor.setText( Integer.toString( valor ) );
79                campoBits.setText( obtenerBits( valor ) );
80             }
81
82          } // fin de la clase interna anónima
83
84       ); // fin de la llamada a addActionListener
85
86       // botón para desplazar el valor una posición a la derecha sin signo
87       JButton botonDerechaCero = new JButton( ">>>" );
88       contenedor.add( botonDerechaCero );
89
90       botonDerechaCero.addActionListener(
91
92          new ActionListener() { // clase interna anónima
93
94             // desplazar una posición a la derecha y mostrar el nuevo valor
95             public void actionPerformed( ActionEvent evento )
96             {
97                int valor = Integer.parseInt( campoValor.getText() );
98                valor >>>= 1;
99                campoValor.setText( Integer.toString( valor ) );
100
101               campoBits.setText( obtenerBits( valor ) );
102            }
103
104         } // fin de la clase interna anónima
105
106      ); // fin de la llamada a addActionListener
107
108      setSize( 400, 120 );
109      setVisible( true );
110
111   } // fin del constructor
112
113   // mostrar representación de bits del valor int especificado
114   private String obtenerBits( int valor )
115   {
116      // crear valor int con 1 en el bit más a la izquierda y 0s en las demás posiciones
117      int mascaraMostrar = 1 << 31;
118
119      StringBuffer bufer = new StringBuffer( 35 ); // búfer para la salida
120
121      // para cada bit anexar 0 o 1 al búfer
122      for ( int bit = 1; bit <= 32; bit++ ) {
123
124         // usar mascaraMostrar para aislar el bit
125         bufer.append( ( valor & mascaraMostrar ) == 0 ? '0' : '1' );
126
127         valor <<= 1; // desplazar el valor una posición a la izquierda
```

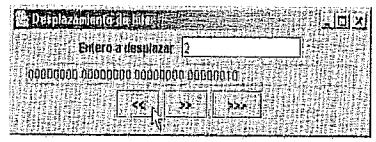
http://librosysolucionarios.net

Figura 21.11 Operadores de desplazamiento a nivel de bits. (Parte 3 de 5.)

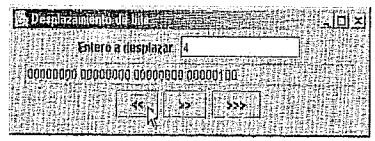

```
128
129            if ( bit % 8 == 0 )
130               bufer.append( ' ' ); // anexar un espacio al búfer cada 8 bits
131         }
132
133         return bufer.toString();
134
135      } // fin del método obtenerBits
136
137      public static void main( String args[] )
138      {
139         DesplazamientoBits aplicacion = new DesplazamientoBits();
140         aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
141      }
142
143   } // fin de la clase DesplazamientoBits
```



Iniciar con el entero 1



Desplazar una posición a la izquierda



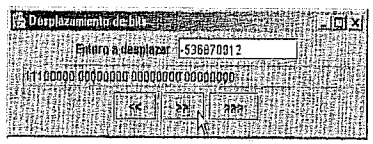
Desplazar una posición a la izquierda



Iniciar con el entero -2147483648



Desplazar una posición a la derecha con signo



Desplazar una posición a la derecha con signo

Figura 21.11 Operadores de desplazamiento a nivel de bits. (Parte 4 de 5.)

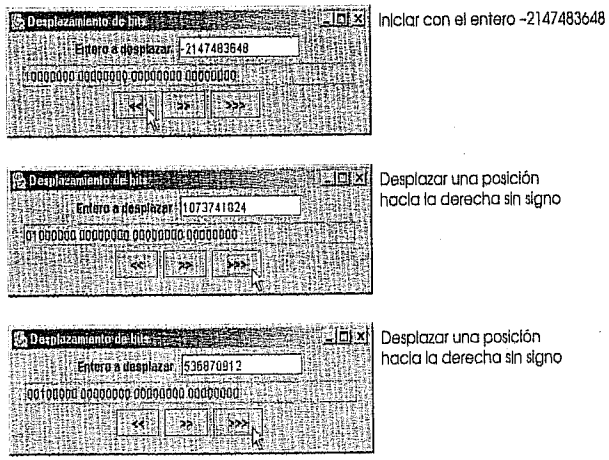


Figura 21.11 Operadores de desplazamiento a nivel de bits. (Parte 5 de 5.)

Operadores de asignación a nivel de bits	
`&=`	Operador de asignación AND a nivel de bits.
`|=`	Operador de asignación OR incluyente a nivel de bits.
`^=`	Operador de asignación OR excluyente a nivel de bits.
`<<=`	Operador de asignación de desplazamiento a la izquierda.
`>>=`	Operador de asignación de desplazamiento a la derecha con signo.
`>>>=`	Operador de asignación de desplazamiento a la derecha sin signo.

Figura 21.12 Operadores de asignación a nivel de bits.

21.7 La clase `BitSet`

La clase ***`BitSet`*** facilita la creación y manipulación de *conjuntos de bits*, los cuales son útiles para representar conjuntos de banderas **`boolean`**. Los objetos **`BitSet`** pueden cambiar su tamaño en forma dinámica; pueden agregarse más bits según sea necesario, y un objeto **`BitSet`** crecerá para alojar los bits adicionales. La clase **`BitSet`** proporciona dos constructores: uno sin argumentos que crea un objeto **`BitSet`** vacío; y otro que recibe un entero, el cual representa el número de bits en el objeto **`BitSet`**. De manera predeterminada, cada bit en un objeto **`BitSet`** tiene un valor **`false`**; el bit subyacente tiene el valor 0. Un bit se establece en **`true`** (lo que también se conoce como "encendido") mediante una llamada al método ***`set`*** de **`BitSet`**, el cual recibe como argumento el índice del bit que se va a poner en 1. Esto hace que el valor subyacente de ese bit sea 1. Observe que los índices de los bits empiezan desde cero, al igual que los arreglos. Un bit se establece en **`false`** (lo que también se conoce como "apagado") llamando al método ***`clear`*** de **`BitSet`**. Esto hace que el valor subyacente de ese bit sea 0. Para obtener el valor de un bit se utiliza el método ***`get`*** de **`BitSet`**, el cual recibe el índice del bit que se va a obtener y devuelve un valor booleano que representa si el bit en ese índice está encendido (**`true`**) o apagado (**`false`**).

La clase **`BitSet`** también proporciona métodos para combinar los bits en dos objetos **`BitSet`**, utilizando los operadores AND lógico (***`and`***), OR incluyente lógico a nivel de bits (***`or`***) y OR excluyente lógico a nivel de bits (***`xor`***). Suponiendo que **`b1`** y **`b2`** son objetos **`BitSet`**, la instrucción:

```
b1.and( b2 );
```

http://librosysolucionarios.net

realiza una operación AND lógica, bit por bit, entre los objetos **`BitSet`** **`b1`** y **`b2`**. El resultado se almacena en **`b1`**. Las operaciones OR incluyente lógico a nivel de bits y XOR lógico a nivel de bits se llevan a cabo mediante las instrucciones:

```
b1.or( b2 );
b1.xor( b2 );
```

El método ***`size`*** de **`BitSet`** devuelve el número de bits en un objeto **`BitSet`**. El método ***`equals`*** de **`BitSet`** compara la igualdad entre dos objetos **`BitSet`**. Dos objetos **`BitSet`** son iguales solamente si cada objeto **`BitSet`** tiene valores idénticos en los bits correspondientes. El método ***`toString`*** de **`BitSet`** crea una representación de cadena del contenido de un objeto **`BitSet`**.

En la figura 21.13 utilizaremos nuevamente la Criba de Eratóstenes (para encontrar números primos), la cual utilizamos en el ejercicio 7.27. En este ejemplo emplearemos un objeto **`BitSet`** en vez de un arreglo para implementar el algoritmo. El programa muestra a todos los números primos desde el 2 hasta el 1023 en un objeto **`JTextArea`** y proporciona un objeto **`JTextField`**, en donde el usuario puede escribir cualquier número, desde el 2 hasta el 1023, para determinar si el número es primo. (El resultado se muestra en un objeto **`JLabel`** en la parte inferior de la ventana.)

```
1  // Fig. 21.13: PruebaBitSet.java
2  // Uso de un objeto BitSet para demostrar la Criba de Eratóstenes.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7
8  public class PruebaBitSet extends JFrame {
9     private BitSet criba;
10    private JLabel etiquetaEstado;
11    private JTextField campoEntrada;
12
13    // configurar la GUI
14    public PruebaBitSet()
15    {
16       super( "Objetos BitSet" );
17
18       criba = new BitSet( 1024 );
19
20       Container contenedor = getContentPane();
21
22       etiquetaEstado = new JLabel( "" );
23       contenedor.add( etiquetaEstado, BorderLayout.SOUTH );
24
25       JPanel panelEntrada = new JPanel();
26       panelEntrada.add( new JLabel( "Escriba un valor entre 2 y 1023" ) );
27
28       // campo de texto para que el usuario introduzca un valor entre 2 y 1023
29       campoEntrada = new JTextField( 10 );
30       panelEntrada.add( campoEntrada );
31       contenedor.add( panelEntrada, BorderLayout.NORTH );
32
33       campoEntrada.addActionListener(
34
35          new ActionListener() { // clase interna
36
37             // determinar si valor es número primo
```

Figura 21.13 La Criba de Eratóstenes, utilizando un objeto **`BitSet`**. (Parte 1 de 3.)

```
38              public void actionPerformed( ActionEvent evento )
39              {
40                 int valor = Integer.parseInt( campoEntrada.getText() );
41
42                 if ( criba.get( valor ) )
43                    etiquetaEstado.setText( valor + " es un número primo" );
44
45                 else
46                    etiquetaEstado.setText( valor + " no es un número primo" );
47              }
48
49           } // fin de la clase interna
50
51        ); // fin de la llamada a addActionListener
52
53        JTextArea areaPrimos = new JTextArea();
54
55        contenedor.add( new JScrollPane( areaPrimos ), BorderLayout.CENTER );
56
57        int tamanio = criba.size(); // establecer todos los bits entre 2 y 1023
58
59        for ( int i = 2; i < tamanio; i++ )
60           criba.set( i );
61
62        // ejecutar el método de la Criba de Eratóstenes
63        int bitFinal = ( int ) Math.sqrt( tamanio);
64
65        for ( int i = 2; i < bitFinal; i++ )
66
67           if ( criba.get( i ) )
68
69              for ( int j = 2 * i; j < tamanio; j += i )
70                 criba.clear( j );
71
72        int contador = 0; // mostrar números primos entre 2 y 1023
73
74        for ( int i = 2; i < tamanio; i++ )
75
76           if ( criba.get( i ) ) {
77              areaPrimos.append( String.valueOf( i ) );
78              areaPrimos.append( ++contador % 7 == 0 ? "\n" : "\t" );
79           }
80
81        setSize( 600, 450 );
82        setVisible( true );
83
84     } // fin del constructor
85
86     public static void main( String args[] )
87     {
88        PruebaBitSet aplicacion = new PruebaBitSet();
89        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
90     }
91
92  } // fin de la clase PruebaBitSet
```

Figura 21.13 La Criba de Eratóstenes, utilizando un objeto **BitSet**. (Parte 2 de 3.)

http://librosysolucionarios.net

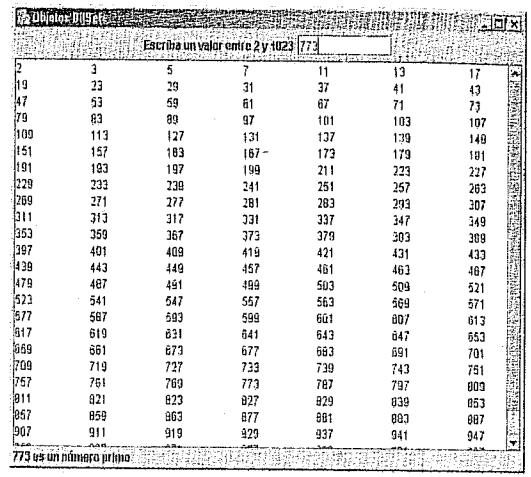


Figura 21.13 La Criba de Eratóstenes, utilizando un objeto **BitSet**. (Parte 3 de 3.)

En la línea 18 se crea un objeto **BitSet** de 1024 bits. En este programa ignoraremos los bits en los índices cero y uno. En las líneas 59 y 60 se establecen todos los bits en el objeto **BitSet** en la posición "encendido", mediante el método **set** de **BitSet**. En las líneas 65 a 70 se determinan todos los números primos del 2 hasta el 1023. El entero **bitFinal** especifica cuándo se completa el algoritmo. El algoritmo básico establece que un número es primo si no tiene otros divisores aparte de 1 y de sí mismo. Empezando con el número 2, una vez que sabemos que un número es primo, podemos eliminar a todos los múltiplos de ese número. El número 2 puede dividirse sólo entre 1 y sí mismo, por lo que es primo. Por lo tanto, podemos eliminar los números 4, 6, 8 y así sucesivamente. La eliminación de un valor consiste en establecer su bit en "apagado" mediante el método **clear** de **BitSet** (línea 70). El número 3 puede dividirse entre 1 y sí mismo. Por lo tanto, podemos eliminar todos los múltiplos de 3. (Tenga en cuenta que todos los números pares ya se han eliminado.) Después de que el programa muestra la lista de números primos, el usuario puede escribir un valor entre 2 y 1023 en el campo de texto y oprimir *Intro* para determinar si el número es primo. El método **actionPerformed** (líneas 38 a 47) utiliza el método **get** de **BitSet** (línea 42) para determinar si está encendido el bit para el número que introdujo el usuario. De ser así, en la línea 43 se muestra un mensaje indicando que el número es primo. En caso contrario, en la línea 46 se muestra un mensaje indicando que el número no es primo.

RESUMEN

- La clase **Vector** maneja arreglos que pueden cambiar su tamaño en forma dinámica. En cualquier momento, un objeto **Vector** contiene un número de elementos que es menor o igual que su capacidad. Si un objeto **Vector** necesita crecer, incrementa su capacidad. Si no se especifica un incremento en la capacidad, Java duplica el tamaño del objeto **Vector** cada vez que se requiera de una capacidad adicional. La capacidad predeterminada es de diez elementos.
- Los objetos **Vector** almacenan referencias a objetos **Object**. Para almacenar valores de tipos primitivos en objetos **Vector**, utilice las clases de tipo de envoltura (**Byte**, **Integer**, **Long**, **Float**, **Double**, **Boolean** y **Character**) para crear objetos que contengan los valores de tipo primitivo.
- La clase **Vector** proporciona cuatro constructores. El constructor sin argumentos crea un objeto **Vector** vacío. El constructor que toma un argumento crea un objeto **Vector** con una capacidad inicial especificada por ese argumento, en caso de que sea **int**, o con base en el número de elementos si es un objeto **Collection**. El constructor que toma dos argumentos crea un objeto **Vector** con una capacidad inicial especificada por el primer argumento y un incremento de capacidad especificado por el segundo argumento.
- El método **add** de **Vector** agrega su argumento al final del objeto **Vector**. El método **insertElementAt** inserta un elemento en la posición especificada. El método **set** establece el elemento en una posición específica.

- El método `remove` de `Vector` elimina la primera ocurrencia de su argumento. El método `removeAllElements` elimina todos los elementos del objeto `Vector`. El método `removeElementAt` elimina el elemento en el índice especificado.
- El método `firstElement` de `Vector` devuelve una referencia al primer elemento. El método `lastElement` devuelve una referencia al último elemento.
- El método `contains` de `Vector` determina si el objeto `Vector` contiene la clave de búsqueda que se especifica como argumento. El método `indexOf` de `Vector` obtiene el índice de la primera ubicación de su argumento. El método devuelve `-1` si el argumento no se encuentra en el objeto `Vector`.
- El método `isEmpty` de `Vector` determina si el objeto `Vector` está vacío. Los métodos `size` y `capacity` determinan el número de elementos que tiene el objeto `Vector` en ese momento y el número de elementos que pueden almacenarse en el objeto `Vector` sin necesidad de asignar más memoria, respectivamente.
- El método `elements` de `Vector` devuelve un objeto `Enumeration` para los elementos de un objeto `Vector`. El método `hasMoreElements` de `Enumeration` determina si hay elementos que no se han visitado todavía para ser enumerados. El método `nextElement` devuelve una referencia al siguiente elemento.
- La clase `Stack` extiende a `Vector`. El método `push` de `Stack` agrega su argumento a la parte superior de la pila. El método `pop` elimina el elemento superior de la pila. El método `peek` devuelve una referencia al elemento superior sin eliminarlo. El método `empty` de `Stack` determina si la pila está vacía.
- Un objeto `Hashtable` asigna claves a los valores.
- La técnica de "hashing" es un esquema de alta velocidad para convertir claves en índices únicos de arreglo, para almacenar y recuperar información.
- El constructor de `Hashtable` sin argumentos crea un objeto `Hashtable` con una capacidad predeterminada de 11 elementos y un factor de carga predeterminado de 0.75. El constructor de `Hashtable` que toma un argumento especifica la capacidad inicial; el constructor que toma dos argumentos especifica la capacidad inicial y el factor de carga, respectivamente.
- El método `put` de `Hashtable` agrega una clave y un valor a un objeto `Hashtable`. El método `get` localiza el valor asociado con la clave especificada. El método `remove` elimina la clave y su valor asociado. El método `isEmpty` determina si la tabla está vacía.
- El método `containsKey` de `Hashtable` determina si la clave especificada como su argumento se encuentra en el objeto `Hashtable`. El método `contains` utiliza el método `equals` de `Object` para determinar si el objeto `Object` especificado como su argumento se encuentra en el objeto `Hashtable`. El método `clear` vacía el objeto `Hashtable`. El método `elements` obtiene un objeto `Enumeration` de los valores. El método `keys` obtiene un objeto `Enumeration` de las claves.
- Un objeto `Properties` es un objeto `Hashtable` persistente. La clase `Properties` extiende a `Hashtable`.
- El constructor de `Properties` sin argumentos crea una tabla `Properties` vacía, sin propiedades predeterminadas. También hay un constructor sobrecargado que recibe una referencia a un objeto `Properties` predeterminado, el cual contiene valores predeterminados para las propiedades.
- El método `setProperty` de `Properties` especifica el valor asociado con la clave que se especifica como argumento. El método `getProperty` de `Properties` localiza el valor de la clave que se especifica como argumento. El método `store` guarda el contenido del objeto `Properties` en el objeto `OutputStream` que se especifica como el primer argumento. El método `load` restaura el contenido del objeto `Properties` que recibe del objeto `InputStream` que se especifica como argumento. El método `propertyNames` obtiene un objeto `Enumeration` de los nombres de las propiedades.
- El operador AND a nivel de bits (`&`) hace que cada bit en el resultado sea `1` si el bit correspondiente en ambos operandos es `1`.
- El operador OR incluyente a nivel de bits (`|`) hace que cada bit en el resultado sea `1` si el bit correspondiente en un operando (o ambos) es `1`.
- El operador OR excluyente a nivel de bits (`^`) hace que cada bit en el resultado sea `1` si el bit correspondiente en sólo uno de los operandos es `1`.
- El operador de desplazamiento a la izquierda (`<<`) desplaza los bits de su operando izquierdo a la izquierda, según el número de bits especificados en su operando derecho.
- El operador de desplazamiento a la derecha con signo (`>>`) desplaza los bits en su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho; si el operando izquierdo es negativo, se desplazan `1`s desde la izquierda; en caso contrario, se desplazan `0`s desde la izquierda.
- El operador de desplazamiento a la derecha sin signo (`>>>`) desplaza los bits de su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho; los `0`s se desplazan desde la izquierda.
- El operador de complemento a nivel de bits (`~`) hace que todos los bits que sean `0` en su operando sean `1` en el resultado y hace que todos los bits que sean `1` en su operando sean `0` en el resultado.

- Cada uno de los operadores a nivel de bits (exceptuando al operador de complemento) tiene su correspondiente operador de asignación.
- El constructor de `BitSet` sin argumentos crea un objeto `BitSet` vacío. El constructor de `BitSet` sin argumentos crea un objeto `BitSet` con el número de bits especificado por su argumento.
- El método `set` de `BitSet` "enciende" el bit especificado. El método `clear` "apaga" el bit especificado. El método `get` devuelve `true` si el bit está encendido, y `false` si el bit está apagado.
- El método `and` de `BitSet` realiza una operación AND lógica, bit por bit, entre dos objetos `BitSet`. El resultado se almacena en el objeto `BitSet` que invocó al método. De manera similar, las operaciones OR lógico a nivel de bits y XOR lógico a nivel de bits se realizan mediante los métodos `or` y `xor`, respectivamente. El método `size` de `BitSet` devuelve el tamaño de un objeto `BitSet`. El método `toString` convierte un objeto `BitSet` en un objeto `String`.

TERMINOLOGÍA

`&` (AND a nivel de bits)
`&=` (AND a nivel de bits)
`^` (OR excluyente a nivel de bits)
`^=` (OR excluyente a nivel de bits)
`|` (OR incluyente a nivel de bits)
`|=` (OR incluyente a nivel de bits)
`~` (complemento a nivel de bits)
`<<` (desplazamiento a la izquierda)
`<<=` (desplazamiento a la izquierda)
`>>` (desplazamiento a la derecha con signo)
`>>=` (desplazamiento a la derecha con signo)
`>>>` (desplazamiento a la derecha sin signo)
`>>>=` (desplazamiento a la derecha sin signo)
`add`, método de la clase `Vector`
`and`, método de la clase `BitSet`
arreglo que puede cambiar su tamaño en forma dinámica
`BitSet`, clase
capacidad de un objeto `Vector`
capacidad inicial de un objeto `Vector`
`capacity`, método de la clase `Vector`
clave en un objeto `Hashtable`
`clear`, método de la clase `BitSet`
colisión en la técnica de "hashing"
conjunto de bits
`contains`, método de la clase `Vector`
`containsKey`, método de la clase `Hashtable`
`elementAt`, método de la clase `Vector`
`elements`, método de la clase `Hashtable`
`elements`, método de la clase `Vector`
`EmptyStackException`, clase
enumerar elementos sucesivos
`Enumeration`, interfaz
`equals`, método de la clase `Object`
factor de carga en la técnica de "hashing"
`firstElement`, método de la clase `Vector`
`get`, método de la clase `BitSet`
`get`, método de la clase `Hashtable`
`getProperty`, método de la clase `Properties`
hashing
`Hashtable`, clase
`hasMoreElements`, método de `Enumeration`
incremento de la capacidad de un objeto `Vector`
`indexOf`, método de la clase `Vector`
`insertElementAt`, método de la clase `Vector`
`isEmpty`, método de la clase `Hashtable`
`isEmpty`, método de la clase `Vector`
iterar a través de los elementos de un contenedor
`java.util`, paquete
`keys`, método de la clase `Hashtable`
`lastElement`, método de la clase `Vector`
`list`, método de la clase `Properties`
`load`, método de la clase `Properties`
`nextElement`, método de `Enumeration`
`NoSuchElementException`, clase
NullPointerException
operadores de asignación a nivel de bits
operadores de manipulación de bits
`or`, método de la clase `BitSet`
par clave/valor
`peek`, método de la clase `Stack`
`pop`, método de la clase `Stack`
`Properties`, clase
`propertyNames`, método de la clase `Properties`
`push`, método de la clase `Stack`
`put`, método de la clase `Hashtable`
`remove`, método de la clase `Hashtable`
`removeAllElements`, método de `Vector`
`removeElement`, método de la clase `Vector`
`removeElementAt`, método de la clase `Vector`
`set`, método de la clase `BitSet`
`set`, método de la clase `Vector`
`setSize`, método de la clase `Vector`
`size`, método de la clase `Hashtable`
`size`, método de la clase `Vector`
`Stack`, clase
`store`, método de la clase `Properties`
tabla de hash persistente
valores predeterminados
`Vector`, clase
`xor`, método de la clase `BitSet`

http://librosysolucionarios.net

EJERCICIOS DE AUTOEVALUACIÓN

21.1 Llene los espacios en blanco en cada uno de los siguientes enunciados:
a) La clase _____ proporciona las capacidades de las estructuras de datos tipo arreglos, que pueden cambiar su propio tamaño en forma dinámica.
b) Si usted no especifica un incremento de capacidad, el sistema _____ el tamaño de un objeto `Vector` cada vez que se requiera de una capacidad adicional.
c) Los bits en el resultado de una expresión que utilice el operador _____ se hacen `1` si los bits correspondientes en ambos operandos son `1`. En caso contrario, los bits se hacen cero.
d) Los bits en el resultado de una expresión que utilice el operador _____ se hacen `1` si cuando menos uno de los bits correspondientes en cualquiera de los operandos es `1`. En caso contrario, los bits se hacen `0`.
e) Los bits en el resultado de una expresión que utilice el operador _____ se hacen `1` si exactamente uno de los bits correspondientes en cualquiera de los operandos es `1`. En caso contrario, los bits se hacen `0`.
f) El operador AND a nivel de bits (`&`) se utiliza a menudo para _____ bits, es decir, para seleccionar ciertos bits de una cadena y establecer otros en `0`.
g) El operador _____ se utiliza para desplazar los bits de un valor dado a la izquierda.
h) El operador _____ desplaza los bits de un valor dado a la derecha con extensión de signo, y el operador _____ desplaza los bits de un valor dado a la derecha con extensión de ceros.

21.2 Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Los valores de tipos primitivos pueden almacenarse directamente en un objeto `Vector`.
b) Con la técnica de "hashing", a medida que aumenta el factor de carga disminuye la probabilidad de que haya colisiones.

21.3 ¿Bajo qué circunstancias se lanza una excepción `EmptyStackException`?

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

21.1 a) `Vector`. b) Duplicará. c) `&`. d) `|`. e) `^`. f) Enmascarar. g) `<<`. h) `>>`, `>>>`.

21.2 a) Falso; un objeto `Vector` solamente almacena objetos `Object`. Un programa debe utilizar las clases de tipos de envoltura (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` y `Character`) del paquete `java.lang` para crear objetos `Object` que contengan los valores de tipos primitivos. b) Falso; a medida que aumenta el factor de carga, hay menos posiciones disponibles en relación con el número total de posiciones, por lo que aumenta la probabilidad de seleccionar una posición ocupada (una colisión) mediante una operación de "hashing".

21.3 Cuando un programa llama a `pop` o `peek` en un objeto `Stack` vacío, se produce una excepción `EmptyStackException`.

EJERCICIOS

21.4 Defina cada uno de los siguientes términos dentro del contexto de "hashing":
a) clave
b) colisión
c) transformación de "hashing"
d) factor de carga
e) concesión entre espacio/tiempo
f) la clase `Hashtable`
g) capacidad de un objeto `Hashtable`

21.5 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `Vector`:
a) `add`
b) `insertElementAt`
c) `set`
d) `remove`
e) `removeAllElements`
f) `removeElementAt`
g) `firstElement`
h) `lastElement`
i) `isEmpty`

j) `contains`
k) `indexOf`
l) `size`
m) `capacity`

21.6 Explique por qué el insertar elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea menor que su capacidad, es una operación relativamente rápida, y por qué el insertar elementos adicionales en un objeto `Vector` cuyo tamaño actual se encuentre al máximo de su capacidad es una operación relativamente lenta.

21.7 Explique el uso de la interfaz `Enumeration` con objetos de la clase `Vector`.

21.8 Al extender la clase `Vector`, los diseñadores de Java pudieron crear la clase `Stack` rápidamente. ¿Cuáles son los aspectos negativos en relación con este uso de la herencia, especialmente para la clase `Stack`?

21.9 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `Hashtable`:
a) `put`
b) `get`
c) `isEmpty`
d) `containsKey`
e) `contains`
f) `keys`

21.10 Utilice un objeto `Hashtable` para crear una clase reutilizable para seleccionar uno de los 13 colores predefinidos en la clase `Color`. Los nombres de los colores deberán utilizarse como claves, y los objetos `Color` predefinidos deberán usarse como valores. Coloque esta clase en un paquete que pueda importarse en cualquier programa de Java. Utilice su nueva clase en una aplicación que permita al usuario seleccionar un color y dibujar una figura en ese color.

21.11 Modifique su solución al ejercicio 14.17 (el programa polimórfico para dibujar) para almacenar todas las figuras que el usuario dibuje en un objeto `Vector` que contenga objetos `MiFigura`. Para los fines de este ejercicio, cree su propia subclase `Vector` llamada `VectorFiguras` que manipule solamente objetos `MiFigura`. Proporcione las siguientes herramientas en su programa:
a) Permitir al usuario del programa eliminar cualquier número de figuras del objeto `Vector` haciendo clic en un botón `Deshacer`.
b) Permitir al usuario seleccionar cualquier figura en la pantalla y desplazarla a una nueva ubicación. Esta operación requiere la adición de un nuevo método a la jerarquía de `MiFigura`. El encabezado del método debe ser:

```
public boolean estaDentro()
```

Este método debe sustituirse para cada subclase de `MiFigura`, para poder determinar si las coordenadas en las que el usuario oprimió el botón del ratón se encuentran dentro de la figura.
c) Permitir al usuario seleccionar cualquier figura en la pantalla y cambiar su color.
d) Permitir al usuario seleccionar cualquier figura en la pantalla que pueda rellenarse o dejarse sin relleno y cambiar su estado de relleno.

21.12 ¿Qué significa cuando decimos que un objeto `Properties` es un objeto `Hashtable` "persistente"? Explique la operación de cada uno de los siguientes métodos de la clase `Properties`:
a) `load`
b) `store`
c) `getProperty`
d) `propertyNames`
e) `list`

21.13 ¿Para qué sería conveniente utilizar objetos de la clase `BitSet`? Explique la operación de cada uno de los siguientes métodos de la clase `BitSet`:
a) `set`
b) `clear`
c) `get`
d) `and`
e) `or`
f) `xor`
g) `size`
h) `equals`
i) `toString`

http://librosysolucionarios.net

21.14 Escriba un programa que desplace una variable entera cuatro bits a la derecha con extensión de signo, y que después desplace la misma variable entera cuatro bits a la derecha con extensión de ceros. El programa deberá imprimir el entero en bits antes y después de cada operación de desplazamiento. Ejecute su programa una vez con un entero positivo, y otra vez con un entero negativo.

21.15 Muestre cómo puede utilizarse el proceso de desplazar un entero a la izquierda una posición para realizar la multiplicación por dos, y cómo puede usarse el proceso de desplazar un entero a la derecha una posición para realizar la división entre dos. No olvide considerar las cuestiones relacionadas con el signo de un entero.

21.16 Escriba un programa que invierta el orden de los bits en un valor entero. El programa deberá recibir el valor como entrada del usuario, y deberá llamar al método `invertirBits` para imprimir los bits en orden inverso. Imprima el valor en bits antes y después de que éstos se inviertan, para confirmar que se hayan invertido correctamente. Tal vez quiera implementar una solución recursiva y una iterativa.

21.17 Modifique su solución al ejercicio 20.10 de manera que utilice la clase `Stack`.

21.18 Modifique su solución al ejercicio 20.12 de manera que utilice la clase `Stack`.

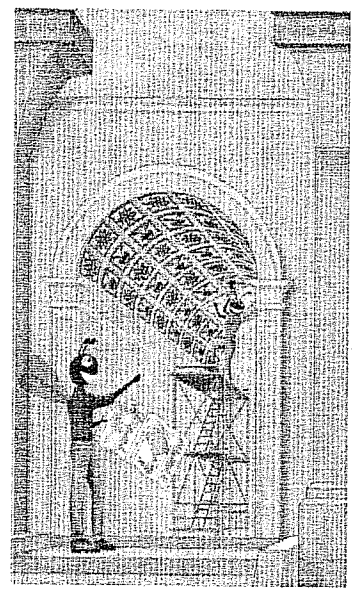
21.19 Modifique su solución al ejercicio 20.13 de manera que utilice la clase `Stack`.

22

Colecciones

Objetivos

- Comprender lo que son las colecciones.
- Utilizar la clase `Arrays` para manipulaciones comunes de arreglos.
- Utilizar las implementaciones del marco de trabajo de colecciones.
- Utilizar los algoritmos del marco de trabajo de colecciones para manipular varias colecciones.
- Utilizar las interfaces del marco de trabajo de colecciones para programar mediante el polimorfismo.
- Utilizar iteradores para "recorrer" los elementos de una colección.
- Comprender las envolturas de sincronización y las envolturas modificables y no modificables.



Creo que ésta es la colección más extraordinaria de talento, de conocimiento humano, que se haya reunido en la Casa Blanca; con la posible excepción de cuando Thomas Jefferson comió solo.
John F. Kennedy

¡Las figuras que puede alojar un contenedor brillante!
Theodore Roethke

Un viaje a través de todo el universo en un mapa.
Miguel de Cervantes

Una ley inmutable en los negocios es que las palabras son palabras,
las explicaciones son explicaciones, las promesas son promesas;
pero sólo el rendimiento es realidad.
Harold S. Green

http://librosysolucionarios.net

Plan general

22.1 Introducción
22.2 Generalidades acerca de las colecciones
22.3 La clase `Arrays`
22.4 La interfaz `Collection` y la clase `Collections`
22.5 Objetos `List`
22.6 Algoritmos
22.6.1 El algoritmo `sort`
22.6.2 El algoritmo `shuffle`
22.6.3 Los algoritmos `reverse`, `fill`, `copy`, `max` y `min`
22.6.4 El algoritmo `binarySearch`
22.7 Conjuntos
22.8 Mapas
22.9 Envolturas de sincronización
22.10 Envolturas no modificables
22.11 Implementaciones abstractas
22.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en el paquete `java.util`
22.12.1 Patrones de diseño de creación
22.12.2 Patrones de diseño de comportamiento
22.12.3 Conclusión
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

22.1 Introducción

En el capítulo 20 vimos cómo crear y manipular estructuras de datos. Aprendimos a crear cada elemento de cada estructura de datos en forma dinámica, y modificamos las estructuras de datos manipulando directamente sus elementos y las referencias a ellos. En este capítulo veremos el *marco de trabajo de colecciones* de Java, el cual contiene estructuras de datos, interfaces y algoritmos preempaquetados para manipular esas estructuras de datos. Algunos ejemplos de colecciones son las tarjetas que usted posee en un juego de cartas, su música favorita almacenada en su computadora y los registros de bienes raíces en el registro de propiedades de su localidad (que asocia números de libro y página con los propietarios de los bienes inmuebles).

Con las colecciones, los programadores utilizan las estructuras de datos existentes sin tener que preocuparse por la manera en que éstas se implementan. Éste es un maravilloso ejemplo de reutilización de código. Los programadores pueden codificar más rápido y esperar un excelente rendimiento, maximizando la velocidad de ejecución y minimizando el consumo de memoria. En este capítulo hablaremos sobre las interfaces del marco de trabajo de colecciones que describen las capacidades de cada tipo de colección, las clases de implementación, los algoritmos que procesan a las colecciones y los *iteradores* que las "recorren".[1]

El marco de trabajo de colecciones de Java proporciona componentes reutilizables, listos para utilizarse;[2] usted no necesita escribir sus propias clases de colecciones. Estas colecciones están estandarizadas, de manera

1. En este capítulo presentamos una introducción al marco de trabajo de colecciones. Para obtener los detalles completos sobre el marco de trabajo de colecciones, visite la página Web `java.sun.com/j2se/1.4.1/docs/guide/collections`.
2. Si usted conoce a C++, estará familiarizado con su marco de trabajo de colecciones, al cual se le conoce como la Biblioteca estándar de plantillas (STL). Vea el capítulo 21 de *Cómo programar en C++, Cuarta edición*, por H. M. Deitel y P. J. Deitel.

©2003, Pearson Educación de México, S.A. de C.V.

que las aplicaciones puedan compartirlas fácilmente sin tener que preocuparse por los detalles relacionados con su implementación. Estas colecciones están escritas para reutilizarse ampliamente; están diseñadas para ejecutarse rápidamente, así como para utilizar la memoria de manera eficiente. El marco de trabajo de colecciones fomenta aún más la reutilización. A medida que se desarrollen estructuras de datos y algoritmos que se ajusten a este marco de trabajo, una extensa base de programadores estará ya familiarizada con las interfaces y algoritmos implementados por esas estructuras de datos.

22.2 Generalidades acerca de las colecciones

Una colección es una estructura de datos (en realidad, un objeto) que puede guardar referencias a otros objetos. Las interfaces de las colecciones declaran las operaciones que un programa puede realizar en cada tipo de colección. Las implementaciones de las colecciones ejecutan las operaciones en ciertas formas, algunas más apropiadas que otras para ciertos tipos de aplicaciones. Las implementaciones de las colecciones se crean cuidadosamente para obtener una ejecución rápida y un uso eficiente de la memoria; además fomentan la reutilización de software al proporcionar una funcionalidad conveniente.

Las interfaces del marco de trabajo de colecciones declaran las operaciones se realizarán en forma genérica, en varios tipos de colecciones. Algunas de las interfaces son `Collection`, `Set`, `List` y `Map`. Dentro del marco de trabajo se proporcionan varias implementaciones de estas interfaces. Los programadores también pueden proporcionar implementaciones que sean específicas a sus propios requerimientos.

El marco de trabajo de colecciones incluye una variedad de características adicionales que minimizan la cantidad de codificación que necesitan realizar los programadores para crear y manipular colecciones.

Las clases e interfaces del marco de trabajo de colecciones son miembros del paquete `java.util`. En la siguiente sección comenzaremos nuestra discusión con un análisis de las herramientas del marco de trabajo de colecciones para manipular arreglos.

22.3 La clase `Arrays`

La clase `Arrays` proporciona métodos estáticos para manipular arreglos. En el capítulo 7, nuestra discusión acerca de la manipulación de arreglos fue de nivel bajo, en el sentido en que escribimos el código en sí para ordenar y buscar en los arreglos. La clase `Arrays` proporciona métodos de alto nivel, como `sort` para ordenar un arreglo, `binarySearch` para buscar en un arreglo ordenado, `equals` para comparar arreglos y `fill` para colocar valores en un arreglo. Estos métodos se sobrecargan para los arreglos de tipo primitivo y los arreglos tipo `Object`. En la figura 22.1 se demuestra el uso de estos métodos.

```
1  // Fig. 22.1: UsoArrays.java
2  // Uso de arreglos en Java.
3  import java.util.*;
4
5  public class UsoArrays {
6     private int valoresInt[] = { 1, 2, 3, 4, 5, 6 };
7     private double valoresDouble[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
8     private int llenoInt[], copiaValoresInt[];
9
10    // inicializar los arreglos
11    public UsoArrays()
12    {
13       llenoInt = new int[ 10 ];
14       copiaValoresInt = new int[ valoresInt.length ];
15
16       Arrays.fill( llenoInt, 7 );      // llenar con 7s
17
18       Arrays.sort( valoresDouble );    // ordenar valoresDouble en forma ascendente
```

Figura 22.1 Métodos de la clase `Arrays`. (Parte 1 de 3.)

http://librosysolucionarios.net

```
19
20          // copiar el arreglo valoresInt en el arreglo copiaValoresInt
21          System.arraycopy( valoresInt, 0, copiaValoresInt,
22             0, valoresInt.length );
23       }
24
25       // mostrar los valores en cada arreglo
26       public void imprimirArreglos()
27       {
28          System.out.print( "valoresDouble: " );
29
30          for ( int cuenta = 0; cuenta < valoresDouble.length; cuenta++ )
31             System.out.print( valoresDouble[ cuenta ] + " " );
32
33          System.out.print( "\nvaloresInt: " );
34
35          for ( int cuenta = 0; cuenta < valoresInt.length; cuenta++ )
36             System.out.print( valoresInt[ cuenta ] + " " );
37
38          System.out.print( "\nllenoInt: " );
39
40          for ( int cuenta = 0; cuenta < llenoInt.length; cuenta++ )
41             System.out.print( llenoInt[ cuenta ] + " " );
42
43          System.out.print( "\ncopiaValoresInt: " );
44
45          for ( int cuenta = 0; cuenta < copiaValoresInt.length; cuenta++ )
46             System.out.print( copiaValoresInt[ cuenta ] + " " );
47
48          System.out.println();
49
50       } // fin del método imprimirArreglos
51
52       // encontrar valor en el arreglo valoresInt
53       public int buscarUnInt( int valor )
54       {
55          return Arrays.binarySearch( valoresInt, valor );
56       }
57
58       // comparar el contenido de los arreglos
59       public void imprimirIgualdad()
60       {
61          boolean b = Arrays.equals( valoresInt, copiaValoresInt );
62
63          System.out.println( "valoresInt " + ( b ? "==" : "!=" ) +
64             " copiaValoresInt" );
65
66          b = Arrays.equals( valoresInt, llenoInt );
67
68          System.out.println( "valoresInt " + ( b ? "==" : "!=" ) +
69             " llenoInt" );
70       }
71
72       public static void main( String args[] )
73       {
```

Figura 22.1 Métodos de la clase **Arrays**. (Parte 2 de 3.)


```
74          UsoArrays usoArrays = new UsoArrays();
75
76          usoArrays.imprimirArreglos();
77          usoArrays.imprimirIgualdad();
78
79          int ubicacion = usoArrays.buscarUnInt( 5 );
80          System.out.println( ( ubicacion >= 0 ? "Se encontro el 5 en el elemento " +
81             ubicacion : "No se encontro el 5" ) + " en valoresInt" );
82
83          ubicacion = usoArrays.buscarUnInt( 8763 );
84          System.out.println( ( ubicacion >= 0 ? "Se encontro el 8763 en el elemento " +
85             ubicacion : "No se encontro el 8763" ) + " en valoresInt" );
86       }
87
88    } // fin de la clase UsoArrays
```

```
valoresDouble: 0.2 3.4 7.9 8.4 9.3
valoresInt: 1 2 3 4 5 6
llenoInt: 7 7 7 7 7 7 7 7 7 7
copiaValoresInt: 1 2 3 4 5 6
valoresInt == copiaValoresInt
valoresInt != llenoInt
Se encontro el 5 en el elemento 4 en valoresInt
No se encontro el 8763 en valoresInt
```

Figura 22.1 Métodos de la clase **Arrays**. (Parte 3 de 3.)

En la línea 16 se hace una llamada al método **fill** de **Arrays** para llenar los 10 elementos del arreglo **llenoInt** con 7s. Las versiones sobrecargadas de **fill** permiten al programador llenar un rango específico de elementos con el mismo valor.

En la línea 18 se ordenan los elementos del arreglo **valoresDouble**. Las versiones sobrecargadas de **sort** permiten al programador ordenar un rango específico de elementos. El método estático **sort** de **Arrays** ordena los elementos del arreglo en forma ascendente, de manera predeterminada. Más adelante en este capítulo veremos cómo ordenar elementos en forma descendente.

En las líneas 21 y 22 se copia el arreglo **valoresInt** en el arreglo **copiaValoresInt**. El primer argumento (**valoresInt**) que se pasa al método ***arraycopy*** de **System** es el arreglo a partir del cual se van a copiar los elementos. El segundo argumento (**0**) es el índice que especifica el punto de inicio en el rango de elementos que se van a copiar del arreglo. Este valor puede ser cualquier índice de arreglo válido. El tercer argumento (**copiaValoresInt**) especifica el arreglo de destino que almacenará la copia. El cuarto argumento (**0**) especifica el índice en el arreglo de destino en donde deberá guardarse el primer elemento copiado. El último argumento especifica el número de elementos a copiar del arreglo en el primer argumento. En este caso copiaremos todos los elementos en el arreglo.

En la línea 55 se hace una llamada al método estático **binarySearch** de **Array** para realizar una búsqueda binaria en **valoresInt**, utilizando **valor** como la clave. Si se encuentra **valor**, **binarySearch** devuelve el índice del elemento. Si no se encuentra **valor**, **binarySearch** devuelve un valor negativo. El valor negativo devuelto se basa en el *punto de inserción* de la clave de búsqueda: el índice en donde se insertaría la clave en el árbol de búsqueda binario si se fuera a realizar una operación de inserción. Una vez que **binarySearch** determina el punto de inserción, cambia el signo de éste a negativo y le resta **1** para obtener el valor de retorno. Por ejemplo, en la figura 22.1 el punto de inserción para el valor **8763** es el elemento en el índice 6. El método **binarySearch** cambia el punto de inserción a **-6**, le resta **1** y devuelve el valor **-7**. Este valor de retorno es útil para agregar elementos en un arreglo ordenado.

http://librosysolucionarios.net

Error común de programación 22.1

Pasar un arreglo desordenado al método `binarySearch` *es un error lógico. El valor devuelto por* `binarySearch` *es indefinido en un caso así.*

En las líneas 61 y 66 se hace una llamada al método `equals` de `Arrays` para determinar si los elementos de dos arreglos son equivalentes. Si los arreglos son iguales, el método devuelve `true`; en caso contrario, devuelve `false`.

Ver un objeto Array como objeto `List`

Una de las características más importantes del marco de trabajo de colecciones es la habilidad de manipular los elementos de un tipo de colección a través de otro tipo de colección, sin importar la implementación interna de esa colección. Al conjunto público de métodos a través del cual se manipulan las colecciones se le conoce como una *vista*.

La clase `Arrays` proporciona el método estático `asList` para ver un arreglo como una colección `List` (la cual encapsula un comportamiento similar al de las listas enlazadas creadas en el capítulo 20; hablaremos más sobre los objetos `List` en la sección 22.5). Una vista `List` permite al programador manipular el arreglo mediante la programación, como si fuera un objeto `List`, mediante llamadas a los métodos de `List`. Cualquier modificación realizada a través de la vista `List` cambia al arreglo, y cualquier modificación realizada a arreglo cambia a la vista `List`. En la figura 22.2 se demuestra el uso del método `asList`.

```
1  // Fig. 22.2: UsoAsList.java
2  // Uso del método asList.
3  import java.util.*;
4
5  public class UsoAsList {
6     private static final String valores[] = { "rojo", "blanco", "azul" };
7     private List lista;
8
9     // inicializar objeto List y establecer valor en ubicación 1
10    public UsoAsList()
11    {
12       lista = Arrays.asList( valores );    // obtener objeto List
13       lista.set( 1, "verde" );             // cambiar un valor
14    }
15
16    // mostrar objeto List y arreglo
17    public void imprimirElementos()
18    {
19       System.out.print( "Elementos del objeto List : " );
20
21       for ( int cuenta = 0; cuenta < lista.size(); cuenta++ )
22          System.out.print( lista.get( cuenta ) + " " );
23
24       System.out.print( "\nElementos del arreglo: " );
25
26       for ( int cuenta = 0; cuenta < valores.length; cuenta++ )
27          System.out.print( valores[ cuenta ] + " " );
28
29       System.out.println();
30    }
31
32    public static void main( String args[] )
33    {
34       new UsoAsList().imprimirElementos();
```

Figura 22.2 Método estático `asList` de `Arrays`. (Parte 1 de 2.)


```
35    }
36
37 } // fin de la clase UsoAsList
```

```
Elementos del objeto List : rojo verde azul
Elementos del arreglo: rojo verde azul
```

Figura 22.2 Método estático `asList` de `Arrays`. (Parte 2 de 2.)

En la línea 7 se declara una referencia `List` llamada `lista`. En la línea 12 se utiliza el método estático `asList` de `Arrays` para obtener una vista `List` de tamaño fijo del arreglo `valores`.

Tip de rendimiento 22.1

`Arrays.asList` crea un objeto `List` de tamaño fijo que opera más rápido que cualquiera de las implementaciones de `List` que se proporcionan.

Error común de programación 22.2

Un objeto `List` creado con `Arrays.asList` tiene un tamaño fijo; si se hacen llamadas a los métodos `add` o `remove` se lanza una excepción `UnsupportedOperationException`.

En la línea 13 se hace una llamada al método `set` de `List` para cambiar el contenido del elemento 1 de `List` a `"verde"`. El programa ve al arreglo como si fuera un objeto `List`, por lo que en la línea 13 se cambia el elemento `valores[1]` del arreglo de `"blanco"` a `"verde"`. Cualquier cambio realizado a la vista `List` se realiza también en el objeto arreglo subyacente.

Observación de ingeniería de software 22.1

Con el marco de trabajo de colecciones, hay muchos métodos que se aplican a objetos `List` y `Collection` que sería conveniente poder utilizar en arreglos. `Arrays.asList` le permite pasar un arreglo a un método que espera un parámetro `List` o `Collection`.

En la línea 21 se hace una llamada al método `size` de `List` para obtener el número de elementos en el objeto `List`. En la línea 22 se hace una llamada al método `get` de `List` para recuperar un elemento individual del objeto `List`. Observe que el valor devuelto por `size` es igual al número de elementos en el arreglo `valores` (es decir, `valores.length`) y que los elementos devueltos por `get` son los elementos del arreglo `valores`.

22.4 La interfaz `Collection` y la clase `Collections`

La interfaz `Collection` es la interfaz raíz en la jerarquía de colecciones, a partir de la cual se derivan las interfaces `Set` (una colección que no contiene duplicados, descrita en la sección 22.7) y `List`. La interfaz `Collection` contiene *operaciones masivas* (es decir, operaciones que se llevan a cabo en toda la colección) para agregar, borrar, comparar y retener objetos (también conocidos como *elementos*) en la colección. Un objeto `Colection` también puede convertirse en un arreglo. Además, la interfaz `Collection` proporciona un método que devuelve un objeto `Iterator`, el cual es similar a un objeto `Enumeration` (presentado en el capítulo 21). Una diferencia es que un objeto `Iterator` puede eliminar elementos de una colección, mientras que un objeto `Enumeration` no. Otros métodos de la interfaz `Collection` permiten a un programa determinar el tamaño de una colección, y si está vacía o no.

Buena práctica de programación 22.1

Un objeto `Iterator` es más flexible que un objeto `Enumeration`, por lo que puede ser conveniente utilizar un objeto `Iterator` cuando sea necesario recorrer un objeto `Collection`.

Observación de ingeniería de software 22.2

`Collection` se utiliza comúnmente con un tipo de parámetro de métodos para permitir el procesamiento polimórfico de todos los objetos que implementen a la interfaz `Collection`.

http://librosysolucionarios.net

Observación de ingeniería de software 22.3

La mayoría de las implementaciones de colecciones proporcionan un constructor que toma un argumento `Collection`*, permitiendo así que un tipo de colección sea tratado como otro tipo de colección.*

La clase `Collections` proporciona métodos `static` que manipulan las colecciones mediante el polimorfismo. Estos métodos implementan algoritmos para buscar, ordenar, etcétera. Usted aprenderá más acerca de estos algoritmos en la sección 22.6. Otros métodos de `Collections` incluyen *métodos de envoltura* que devuelven nuevas colecciones. Hablaremos sobre los métodos de envoltura en las secciones 22.9 y 22.10.

22.5 Objetos `List`

Un objeto `List` (conocido como *secuencia*) es un objeto `Collection` ordenado que puede contener elementos duplicados. Al igual que los índices de arreglos, los índices de objetos `List` empiezan desde cero (es decir, el índice del primer elemento es cero). Además de los métodos de interfaz heredados de `Collection`, `List` proporciona métodos para manipular elementos a través de sus índices, para manipular un rango especificado de elementos, buscando elementos y para obtener un objeto `ListIterator` para acceder a los elementos.

La interfaz `List` es implementada por varias clases, incluyendo a `ArrayList`, `LinkedList` y `Vector`. La clase `ArrayList` es una implementación de un objeto `List` como arreglo que puede modificar su tamaño. El comportamiento y las herramientas de la clase `ArrayList` son similares a las de la clase `Vector`, presentada en el capítulo 21. Un objeto `LinkedList` es una implementación de un objeto `List` como lista enlazada.

Tip de rendimiento 22.2

Los objetos `ArrayList` *se comportan igual que los objetos* `Vector` *desincronizados, y por lo tanto se ejecutan con más rapidez que los objetos* `Vector`*, ya que los objetos* `ArrayList` *no tienen la sobrecarga que implica la sincronización de los procesos.*

Observación de ingeniería de software 22.4

Los objetos `LinkedList` *pueden usarse para crear pilas, colas, árboles y colas con dos partes finales (conocidas en inglés como "deque").*

```
1  // Fig. 22.3: PruebaCollection.java
2  // Uso de la interfaz Collection.
3  import java.awt.Color;
4  import java.util.*;
5
6  public class PruebaCollection {
7     private static final String colores[] = { "rojo", "blanco", "azul" };
8
9     // crear objeto ArrayList, agregarle objetos y manipularlo
10    public PruebaCollection()
11    {
12       List lista = new ArrayList();
13
14       // agregar objetos a lista
15       lista.add( Color.MAGENTA );      // agregar un objeto color
16
17       for ( int cuenta = 0; cuenta < colores.length; cuenta++ )
18          lista.add( colores[ cuenta ] );
19
20       lista.add( Color.CYAN );         // agregar un objeto color
21
22       // mostrar contenido de lista
23       System.out.println( "\nArrayList: " );
```

Figura 22.3 Demostración de la interfaz `Collection` mediante un objeto `ArrayList`. (Parte 1 de 2.)


```
24
25       for ( int cuenta = 0; cuenta < lista.size(); cuenta++ )
26          System.out.print( lista.get( cuenta ) + " " );
27
28       // eliminar todos los objetos String
29       eliminarObjetosString( lista );
30
31       // mostrar contenido de lista
32       System.out.println( "\n\nArrayList despues de llamar a eliminarObjetosString: " );
33
34       for ( int cuenta = 0; cuenta < lista.size(); cuenta++ )
35          System.out.print( lista.get( cuenta ) + " " );
36
37    } // fin del constructor PruebaCollection
38
39    // eliminar objetos String del objeto Collection
40    private void eliminarObjetosString( Collection coleccion )
41    {
42       Iterator iterador = coleccion.iterator(); // obtener iterador
43
44       // iterar mientras coleccion tenga elementos
45       while ( iterador.hasNext() )
46
47          if ( iterador.next() instanceof String )
48             iterador.remove();  // eliminar objeto String
49    }
50
51    public static void main( String args[] )
52    {
53       new PruebaCollection();
54    }
55
56 } // fin de la clase PruebaCollection
```

```
ArrayList:
java.awt.Color[r=255,g=0,b=255] rojo blanco azul java.awt.Color
[r=0,g=255,b=255]

ArrayList despues de llamar a eliminarObjetosString:
java.awt.Color[r=255,g=0,b=255] java.awt.Color[r=0,g=255,b=255]
```

Figura 22.3 Demostración de la interfaz `Collection` mediante un objeto `ArrayList`. (Parte 2 de 2.)

En la figura 22.3 se utiliza un objeto `ArrayList` para demostrar varias herramientas de la interfaz `Collection`. El programa coloca objetos `String` y `Color` en un objeto `ArrayList` y utiliza un objeto `Iterator` para eliminar los objetos `String` de la colección `ArrayList`.

En la línea 12 se crea una instancia de un objeto `ArrayList` y se asigna su referencia a la variable `lista`. En las líneas 15 a 20 se llena la variable `lista` con objetos `Color` y `String`. En las líneas 23 a 26 se muestra cada elemento de `lista`. En la línea 25 se hace una llamada al método `size` de `List` para obtener el número de elementos en el objeto `ArrayList`. En la línea 26 se utiliza el método `get` de `List` para recuperar valores de elementos individuales. En la línea 29 se hace una llamada al método `eliminarObjetosString` (declarado en las líneas 40 a 49) y se le pasa `lista` como argumento. El método `eliminarObjetosString` elimina objetos `String` de una colección. En las líneas 32 a 35 se imprimen los elementos de `lista` después de que `eliminarObjetosString` elimina los objetos `String` de la lista. La salida de la figura 22.3 contiene solamente objetos `Color`, una vez que se eliminan los objetos `String`.

http://librosysolucionarios.net

El método **eliminarObjetosString** declara un parámetro de tipo **Collection** (línea 40) que permite pasar cualquier objeto **Collection** como argumento para este método. El método accede a los elementos del objeto **Collection** mediante un objeto **Iterator**. La línea 42 llama al método ***iterator*** de **Collection**, el cual obtiene un objeto **Iterator** para el objeto **Collection**. En la condición del ciclo (línea 45) se hace una llamada al método ***hasNext*** de **Iterator** para determinar si el objeto **Collection** contiene más elementos. El método **hasNext** devuelve **true** si otro elemento existe, y devuelve **false** en caso contrario.

La condición del **if** en la línea 47 hace una llamada al método ***next*** de **Iterator** para obtener una referencia al siguiente elemento, después utiliza **instanceof** para determinar si el objeto es de tipo **String**. De ser así, en la línea 48 se hace una llamada al método ***remove*** de **Iterator** para eliminar el objeto **String** del objeto **Collection**.

Error común de programación 22.3



Al iterar a través de una colección con un objeto **Iterator**, *use el método* ***remove*** *de* **Iterator** *para eliminar un elemento de la colección. Los iteradores son de "falla rápida"; si la colección es modificada por uno de los métodos de la colección después de que se crea un iterador para esa colección, cada operación del iterador lanzará una excepción* **ConcurrentModificationException**.

En la figura 22.4 se muestran operaciones en objetos **LinkedList**. El programa crea dos objetos **LinkedList**, en donde cada uno de ellos contiene objetos **String**. Los elementos de un objeto **List** se agregan al otro. Después todos los objetos **String** se convierten a mayúsculas y se elimina un rango de elementos.

En las líneas 14 y 15 se crean los objetos **LinkedList** llamados **enlace** y **enlace2**, respectivamente. En las líneas 18 a 21 se hace una llamada al método **add** de **List** para anexar elementos de los arreglos **colores** y **colores2** al final de **enlace** y **enlace2**, respectivamente.

```
1  // Fig. 22.4: PruebaList.java
2  // Uso de objetos LinkedList.
3  import java.util.*;
4
5  public class PruebaList {
6     private static final String colores[] = { "negro", "amarillo",
7        "verde", "azul", "violeta", "plata" };
8     private static final String colores2[] = { "oro", "blanco",
9        "cafe", "azul", "gris", "plata" };
10
11    // establecer y manipular los objetos LinkedList
12    public PruebaList()
13    {
14       List enlace = new LinkedList();
15       List enlace2 = new LinkedList();
16
17       // agregar elementos a cada lista
18       for ( int cuenta = 0; cuenta < colores.length; cuenta++ ) {
19          enlace.add( colores[ cuenta ] );
20          enlace2.add( colores2[ cuenta ] );
21       }
22
23       enlace.addAll( enlace2 );              // concatenar listas
24       enlace2 = null;                        // liberar recursos
25
26       imprimirLista( enlace );
27
```

Figura 22.4 Objetos **List** y **ListIterator**. (Parte 1 de 3.)


```
28       objetosStringMayuscula( enlace );
29
30       imprimirLista( enlace );
31
32       System.out.print( "\nEliminando elementos 4 a 6..." );
33       eliminarElementos( enlace, 4, 7 );
34
35       imprimirLista( enlace );
36
37       imprimirListaInversa( enlace );
38
39    } // fin del constructor de PruebaList
40
41    // mostrar contenido de objeto List
42    public void imprimirLista( List lista )
43    {
44       System.out.println( "\nlista: " );
45
46       for ( int cuenta = 0; cuenta < lista.size(); cuenta++ )
47          System.out.print( lista.get( cuenta ) + " " );
48
49       System.out.println();
50    }
51
52    // localizar objetos String y convertir en mayúscula
53    private void objetosStringMayuscula( List lista )
54    {
55       ListIterator iterador = lista.listIterator();
56
57       while ( iterador.hasNext() ) {
58          Object objeto = iterador.next(); // obtener elemento
59
60          if ( objeto instanceof String )  // verificar si es String
61             iterador.set( ( String ) objeto ).toUpperCase() );
62       }
63    }
64
65    // obtener sublista y utilizar método clear para eliminar elementos de sublista
66    private void eliminarElementos( List lista, int inicio, int fin )
67    {
68       lista.subList( inicio, fin ).clear(); // eliminar elementos
69    }
70
71    // imprimir lista inversa
72    private void imprimirListaInversa( List lista )
73    {
74       ListIterator iterador = lista.listIterator( lista.size() );
75
76       System.out.println( "\nLista inversa:" );
77
78       // imprimir lista en orden inverso
79       while( iterador.hasPrevious() )
80          System.out.print( iterador.previous() + " " );
81    }
82
```

Figura 22.4 Objetos **List** y **ListIterator**. (Parte 2 de 3.)

http://librosysolucionarios.net

```
83     public static void main( String args[] )
84     {
85        new PruebaList();
86     }
87
88  } // fin de la clase PruebaList
```

```
lista:
negro amarillo verde azul violeta plata oro blanco cafe azul gris plata

lista:
NEGRO AMARILLO VERDE AZUL VIOLETA PLATA ORO BLANCO CAFE AZUL GRIS PLATA

Eliminando elementos 4 a 6...
lista:
NEGRO AMARILLO VERDE AZUL BLANCO CAFE AZUL GRIS PLATA

Lista inversa:
PLATA GRIS AZUL CAFE BLANCO AZUL VERDE AMARILLO NEGRO
```

Figura 22.4 Objetos `List` y `ListIterator`. (Parte 3 de 3.)

En la línea 23 se hace una llamada al método ***addAll*** de `List` para anexar todos los elementos de `enlace2` al final de `enlace`. En la línea 24 se establece `enlace2` en `null`, de manera que el objeto `LinkedList` al que hacía referencia `enlace2` pueda marcarse para la recolección de basura. En la línea 26 se hace una llamada al método `imprimirLista` (líneas 42 a 50) para mostrar el contenido de la lista `enlace`. En la línea 28 se hace una llamada al método `objetosStringMayuscula` (líneas 53 a 63) para convertir cada elemento `String` en mayúsculas; después en la línea 30 se hace una llamada nuevamente a `imprimirLista` para mostrar los objetos `String` modificados. En la línea 33 se hace una llamada al método `eliminarElementos` (líneas 66 a 69) para eliminar los elementos empezando desde el índice 4 hasta, pero no incluyendo, al índice 7 de la lista. En la línea 37 se hace una llamada al método `imprimirListaInversa` (líneas 72 a 81) para imprimir la lista en orden inverso.

El método `objetosStringMayuscula` (líneas 52 a 63) cambia los elementos `String` en minúscula del argumento `List` por objetos `String` en mayúscula. En la línea 55 se hace una llamada al método ***listIterator*** de `List` para obtener un *iterador bidireccional* (es decir, un iterador que pueda recorrer un objeto `Lista` hacia delante o hacia atrás) para el objeto `List`. En la condición del ciclo `while` (línea 57) se hace una llamada al método `hasNext` para determinar si el objeto `List` contiene otro elemento. En la línea 58 se obtiene el siguiente objeto `Object` en el objeto `List` y se le asigna su referencia a `objeto`. En la línea 60 se determina si `objeto` apunta a un objeto `String`. De ser así, en la línea 61 se convierte `objeto` en `String`, se hace la llamada al método `toUpperCase` de `String` para obtener una versión en mayúsculas del objeto `String` y se hace una llamada al método ***set*** de `Iterator` para reemplazar el objeto `String` actual al que hace referencia `iterador` con el objeto `String` devuelto por el método `toUpperCase`.

El método `eliminarElementos` (líneas 66 a 69) elimina un rango de elementos de la lista. En la línea 68 se hace una llamada al método ***subList*** de `List` para obtener una porción del objeto `List` (lo que se conoce como *sublista*). La sublista es simplemente una vista hacia el interior del objeto `List` desde el que se hace la llamada a `subList`. El método `subList` toma dos argumentos: el índice inicial para la sublista y el índice final. Observe que el índice final no forma parte del rango de la sublista. En este ejemplo, pasamos 4 para el índice inicial y 7 para el índice final a `subList`. La sublista devuelta es el conjunto de elementos con los índices 4 a 6. A continuación, el programa hace una llamada al método ***clear*** de `List` en la sublista para eliminar los elementos que ésta contiene del objeto `List`. Cualquier cambio realizado a una sublista se hace realmente en el objeto `List` original.

El método `imprimirListaInversa` (líneas 72 a 81) imprime la lista al revés. En la línea 74 se hace una llamada al método `listIterator` de `List` con un argumento que especifica la posición inicial (en

http://librosysolucionarios.net

nuestro caso, el último elemento en la lista) para obtener un iterador bidireccional para la lista. En la condición del ciclo `while` (línea 79) se hace una llamada al método `hasPrevious` para determinar si hay más elementos mientras se recorre la lista hacia atrás. En la línea 80 se obtiene el objeto `Object` anterior de la lista y se envía como salida al flujo de salida estándar.

En la figura 22.5 se utiliza el método ***toArray*** para obtener un arreglo de una colección `LinkedList`. El programa agrega una serie de cadenas a un objeto `LinkedList` y hace una llamada al método `toArray` para obtener un arreglo que contiene referencias a esas cadenas.

```
1  // Fig. 22.5: UsoToArray.java
2  // Uso del método toArray.
3  import java.util.*;
4
5  public class UsoToArray {
6
7     // crear objeto LinkedList, agregar elementos y convertir en arreglo
8     public UsoToArray()
9     {
10       String colores[] = { "negro", "azul", "amarillo" };
11
12       LinkedList enlaces = new LinkedList( Arrays.asList( colores ) );
13
14       enlaces.addLast( "rojo" );  // agregar como último elemento
15       enlaces.add( "rosa" );      // agregar al final
16       enlaces.add( 3, "verde" );  // agregar en el 3er índice
17       enlaces.addFirst( "cyan" ); // agregar como primer elemento
18
19       // obtener elementos de objeto LinkedList como un arreglo
20       colores = ( String [] ) enlaces.toArray( new String[ enlaces.size() ] );
21
22       System.out.println( "colores: " );
23
24       for ( int cuenta = 0; cuenta < colores.length; cuenta++ )
25          System.out.println( colores[ cuenta ] );
26    }
27
28    public static void main( String args[] )
29    {
30       new UsoToArray();
31    }
32
33 } // fin de la clase UsoToArray
```

```
colores:
cyan
negro
azul
amarillo
verde
rojo
rosa
```

Figura 22.5 El método `toArray` de `List`.

En la línea 12 se crea un objeto `LinkedList` que contiene los elementos del arreglo `colores` y se asigna la referencia `LinkedList` a `enlaces`. Observe el uso del método `asList` de `Arrays` para inicializar el objeto `LinkedList` con un objeto `Collection`. En la línea 14 se hace una llamada al método ***`addLast`*** para agregar `"rojo"` al final de `enlaces`. En las líneas 15 y 16 se hace una llamada al método ***`add`*** para agregar `"rosa"` como el último elemento y `"verde"` como el elemento en el índice 3 (es decir, el cuarto elemento). En la línea 17 se hace una llamada a ***`addFirst`*** para agregar `"cyan"` como el nuevo primer elemento en el objeto `LinkedList`. (*Nota:* Cuando se agrega `"cyan"` como el primer elemento, `"verde"` se convierte en el quinto elemento en el objeto `LinkedList`.)

En la línea 20 se hace una llamada al método `toArray` de `List` para obtener un arreglo `String` de `enlaces`. El arreglo es una copia de los elementos de la lista; si se modifica el contenido del arreglo no se modifica la lista. El arreglo que se pasa al método `toArray` debe ser del mismo tipo que se desee que devuelva el método `toArray`. Si el número de elementos en el arreglo es mayor que el número de elementos en el objeto `LinkedList`, `toArray` copia los elementos de la lista en su argumento tipo arreglo y devuelve ese arreglo. Si el objeto `LinkedList` tiene más elementos que el número de elementos en el arreglo que se pasa a `toArray`, este método asigna un nuevo arreglo del mismo tipo que recibe como argumento, copia los elementos de la lista en el nuevo arreglo y devuelve a este nuevo arreglo.



Error común de programación 22.4

Pasar un arreglo que contenga datos al método `toArray` puede crear errores lógicos. Si el número de elementos en el arreglo es menor que el número de elementos en el objeto que llama a `toArray`, se asigna más memoria para almacenar los elementos del objeto (sin preservar los elementos del arreglo). Si el número de elementos en el arreglo es mayor que el número de elementos en el objeto, los elementos del arreglo (empezando en el índice cero) se sobrescriben con los elementos del objeto. Los elementos de arreglos que no se sobrescriben retienen sus valores.

22.6 Algoritmos

El marco de trabajo de colecciones proporciona varios algoritmos de alto rendimiento para manipular elementos de colecciones. Estos algoritmos se implementan como métodos estáticos de la clase `Collections`. Los algoritmos ***`sort`***, ***`binarySearch`***, ***`reverse`***, ***`shuffle`***, ***`fill`*** y ***`copy`*** operan en objetos `List`. Los algoritmos ***`min`*** y ***`max`*** operan en objetos `Collection`.

El algoritmo ***`reverse`*** invierte los elementos de un objeto `List`; ***`fill`*** establece cada elemento de un objeto `List` para que haga referencia a un objeto `Object` especificado; y `copy` copia las referencias de un objeto `List` a otro.



Observación de ingeniería de software 22.5

Los algoritmos del marco de trabajo de colecciones son polimórficos. Es decir, cada algoritmo puede operar en objetos que implementen interfaces específicas, sin importar sus implementaciones subyacentes.

22.6.1 El algoritmo `sort`

El algoritmo `sort` ordena los elementos de un objeto `List`. El orden se determina en base al orden natural del tipo de los elementos, según su implementación mediante el método `compareTo` de ese objeto. El método `compareTo` está declarado en la interfaz ***`Comparable`*** y algunas veces se le conoce como el *método de comparación natural*. La llamada a `sort` puede especificar un segundo argumento tipo objeto ***`Comparator`***, para determinar un ordenamiento alterno de los elementos.

El algoritmo `sort` utiliza un *ordenamiento estable*: un ordenamiento que no reordena los elementos equivalentes. El algoritmo `sort` es rápido. Para los lectores que hayan estudiado algo sobre la teoría de la complejidad en cursos sobre estructuras de datos o algoritmos, este ordenamiento se lleva a cabo en un tiempo determinado por $n \log(n)$. (Los lectores que no estén familiarizados con la teoría de la complejidad pueden estar seguros que este algoritmo es extremadamente rápido.)



Observación de ingeniería de software 22.6

La documentación de las APIs de Java algunas veces proporciona los detalles de implementación. Por ejemplo, `sort` se implementa como un ordenamiento de fusión modificado. Evite escribir código que sea dependiente de estos detalles de implementación, ya que éstos pueden cambiar.

http://librosysolucionarios.net

En la figura 22.6 se utiliza el algoritmo `sort` para ordenar los elementos de un objeto `List` en orden ascendente (línea 18). Observe que en las líneas 16 y 21 se utiliza una llamada implícita al método `toString` de la lista para mostrar el contenido de la lista en el formato que se muestra en la segunda y cuarta línea de la salida.

En la figura 22.7 se ordena la misma lista de cadenas utilizadas en la figura 22.6, pero ahora en orden descendente. En este ejemplo se introduce el objeto `Comparator`, el cual se utiliza para ordenar los elementos de un objeto `Collection` en distinto orden.

```
1  // Fig. 22.6: Ordenar1.java
2  // Uso del algoritmo sort.
3  import java.util.*;
4
5  public class Ordenar1 {
6     private static final String palos[] =
7        { "Corazones", "Diamantes", "Treboles", "Espadas" };
8
9     // mostrar los elementos del arreglo
10    public void imprimirElementos()
11    {
12       // crear objeto ArrayList
13       List lista = new ArrayList( Arrays.asList( palos ) );
14
15       // mostrar lista
16       System.out.println( "Elementos desordenados en el arreglo:\n" + lista );
17
18       Collections.sort( lista ); // ordenar objeto ArrayList
19
20       // mostrar lista
21       System.out.println( "Elementos ordenados en el arreglo:\n" + lista );
22    }
23
24    public static void main( String args[] )
25    {
26       new Ordenar1().imprimirElementos();
27    }
28
29 } // fin de la clase Ordenar1
```

```
Elementos desordenados en el arreglo:
[Corazones, Diamantes, Treboles, Espadas]
Elementos ordenados en el arreglo:
[Corazones, Diamantes, Espadas, Treboles]
```

Figura 22.6 El método `sort` de `Collections`.

```
1  // Fig. 22.7: Ordenar2.java
2  // Uso de un objeto Comparator con el algoritmo sort.
3  import java.util.*;
4
```

Figura 22.7 El método `sort` de `Collections` con un objeto `Comparator`. (Parte 1 de 2.)

```
5  public class Ordenar2 {
6     private static final String palos[] =
7        { "Corazones", "Diamantes", "Treboles", "Espadas" };
8
9     // mostrar los elementos del objeto List
10    public void imprimirElementos()
11    {
12       List lista = Arrays.asList( palos ); // crear objeto List
13
14       // mostrar elementos del objeto List
15       System.out.println( "Elementos desordenados en la lista:\n" + lista );
16
17       // ordenar en forma descendente utilizando un comparador
18       Collections.sort( lista, Collections.reverseOrder() );
19
20       // mostrar los elementos del objeto List
21       System.out.println( "Elementos ordenados en la lista:\n" + lista );
22    }
23
24    public static void main( String args[] )
25    {
26       new Ordenar2().imprimirElementos();
27    }
28
29 } // fin de la clase Ordenar2
```

```
Elementos desordenados en la lista:
[Corazones, Diamantes, Treboles, Espadas]
Elementos ordenados en la lista:
[Treboles, Espadas, Diamantes, Corazones]
```

Figura 22.7 El método **sort** de **Collections** con un objeto **Comparator**. (Parte 2 de 2.)

En la línea 18 se hace una llamada al método **sort** de **Collections** para ordenar el objeto **List** en forma descendente. El método ***reverseOrder*** de **Collections** devuelve un objeto **Comparator**, el cual representa el orden inverso de la colección. Para ordenar una vista **List** de un arreglo **String**, el orden inverso es una *comparación lexicográfica* (el comparador compara los valores Unicode que representan a cada elemento) en orden descendente. Es posible crear un objeto **Comparator** personalizado si se define una clase que implemente a la interfaz **Comparator**.

En la figura 22.8 se crea un objeto **Comparator** personalizado. Este programa crea cinco objetos **Tiempo2** y declara una clase **ComparadorTiempo** personalizada para comparar los objetos.

```
1  // Fig. 22.8: Ordenar3.java
2  // Creación de una clase Comparator personalizada.
3  import java.util.*;
4
5  public class Ordenar3 {
6
7     public void imprimirElementos()
```

Figura 22.8 El método **sort** de **Collections** con un objeto **Comparator** personalizado. (Parte 1 de 3.)

http://librosysolucionarios.net


```
8     {
9        List lista = new ArrayList(); //Crear lista
10
11       lista.add( new Tiempo2(  6, 24, 34 ) );
12       lista.add( new Tiempo2( 18, 14, 05 ) );
13       lista.add( new Tiempo2( 12, 07, 58 ) );
14       lista.add( new Tiempo2(  6, 14, 22 ) );
15       lista.add( new Tiempo2(  8, 05, 00 ) );
16
17       // mostrar los elementos del objeto List
18       System.out.println( "Elementos desordenados en la lista:\n" + lista );
19
20       // ordenar utilizando un comparador
21       Collections.sort( lista, new ComparadorTiempo() );
22
23       // mostrar los elementos del objeto List
24       System.out.println( "Elementos ordenados en la lista:\n" + lista );
25    }
26
27    public static void main( String args[] )
28    {
29       new Ordenar3().imprimirElementos();
30    }
31
32    private class ComparadorTiempo implements Comparator {
33       int compararHora, compararMinuto, compararSegundo;
34       Tiempo2 tiempo1, tiempo2;
35
36       public int compare( Object objeto1, Object objeto2 )
37       {
38          // convertir los objetos
39          tiempo1 = ( Tiempo2 ) objeto1;
40          tiempo2 = ( Tiempo2 ) objeto2;
41
42          compararHora = new Integer( tiempo1.obtenerHora() ).compareTo(
43                         new Integer( tiempo2.obtenerHora() ) );
44
45          // probar la hora primero
46          if ( compararHora != 0 )
47             return compararHora;
48
49          compararMinuto = new Integer( tiempo1.obtenerMinuto() ).compareTo(
50                           new Integer( tiempo2.obtenerMinuto() ) );
51
52          // después probar el minuto
53          if ( compararMinuto != 0 )
54             return compararMinuto;
55
56          compararSegundo = new Integer( tiempo1.obtenerSegundo() ).compareTo(
57                            new Integer( tiempo2.obtenerSegundo() ) );
58
59          return compararSegundo; // devolver el resultado de comparar los segundos
60       }
61
```

Figura 22.8 El método **sort** de **Collections** con un objeto **Comparator** personalizado. (Parte 2 de 3.)

```
62     } // fin de la clase ComparadorTiempo
63
64  } // fin de la clase Ordenar3
```

```
Elementos desordenados en la lista:
[06:24:34, 18:14:05, 12:07:58, 06:14:22, 08:05:00]
Elementos ordenados en la lista:
[06:14:22, 06:24:34, 08:05:00, 12:07:58, 18:14:05]
```

Figura 22.8 El método **sort** de **Collections** con un objeto **Comparator** personalizado. (Parte 3 de 3.)

En la línea 9 se crea un objeto **ArrayList**. En las líneas 11 a 15 se crean cinco objetos **Tiempo2** y se agregan a esta lista. En la línea 21 se hace una llamada al método **sort**, en donde se le pasa un objeto de nuestra clase **ComparadorTiempo**.

En las líneas 32 a 62 se declara la clase **ComparadorTiempo**, la cual implementa a la interfaz **Comparator**. El método **compare** (líneas 36 a 60) realiza comparaciones entre objetos **Tiempo2**. En las líneas 39 y 40 se asignan referencias **Tiempo2** a los dos objetos en las variables **tiempo1** y **tiempo2**. En las líneas 42 y 43 se comparan las dos horas de los objetos **Tiempo2**, utilizando el método **compareTo** de **Integer**. Si las horas son distintas (línea 45), entonces devolvemos este valor. Si este valor es positivo, entonces la primera hora es mayor que la segunda y el primer tiempo es mayor que el segundo. Si este valor es negativo, entonces la primera hora es menor que la segunda y el primer tiempo es menor que el segundo. Si este valor es cero, las horas son iguales y debemos comparar los minutos (y tal vez los segundos) para poder determinar cuál tiempo es mayor.

22.6.2 El algoritmo shuffle

El algoritmo **shuffle** ordena al azar los elementos de un objeto **List**. En el capítulo 11 presentamos una simulación para barajar y repartir cartas, en la que se utiliza un ciclo para barajar un mazo de cartas. En la figura 22.9, utilizamos el algoritmo **shuffle** para barajar el mazo de cartas. La mayor parte del código es similar al de la figura 11.19. La acción de barajar las cartas se lleva a cabo en la línea 59, la cual llama al método estático **shuffle** de **Collections** para revolver el arreglo a través de la vista **List** del mismo.

```
1  // Fig. 22.9: Cartas.java
2  // Uso del algoritmo shuffle.
3  import java.util.*;
4
5  // clase para representar un objeto Carta en un mazo de cartas
6  class Carta {
7     private String cara;
8     private String palo;
9
10    // inicializar un objeto Carta
11    public Carta( String caraInicial, String paloInicial )
12    {
13       cara = caraInicial;
14       palo = paloInicial;
```

Figura 22.9 Ejemplo para barajar y repartir cartas mediante el método **shuffle** de **Collections**. (Parte 1 de 3.)

http://librosysolucionarios.net


```
15    }
16
17    // devolver cara de objeto Carta
18    public String obtenerCara()
19    {
20       return cara;
21    }
22
23    // devolver palo de objeto Carta
24    public String obtenerPalo()
25    {
26       return palo;
27    }
28
29    // devolver representación String de objeto Carta
30    public String toString()
31    {
32       StringBuffer bufer = new StringBuffer( cara + " de " + palo );
33       bufer.setLength( 20 );
34
35       return bufer.toString();
36    }
37
38 } // fin de la clase Carta
39
40 // declaración de la clase Cartas
41 public class Cartas {
42    private static final String palos[] =
43       { "Corazones", "Treboles", "Diamantes", "Espadas" };
44    private static final String caras[] = { "As", "Dos", "Tres",
45       "Cuatro", "Cinco", "Seis", "Siete", "Ocho", "Nueve", "Diez",
46       "Joto", "Quina", "Rey" };
47    private List lista;
48
49    // configurar mazo de Cartas y barajar
50    public Cartas()
51    {
52       Carta mazo[] = new Carta[ 52 ];
53
54       for ( int cuenta = 0; cuenta < mazo.length; cuenta++ )
55          mazo[ cuenta ] = new Carta( caras[ cuenta % 13 ],
56             palos[ cuenta / 13 ] );
57
58       lista = Arrays.asList( mazo );  // obtener objeto List
59       Collections.shuffle( lista );  // barajar mazo
60    }
61
62    // mostrar mazo
63    public void imprimirCartas()
64    {
65       int mitad = lista.size() / 2 - 1;
66
67       for ( int i = 0, j = mitad + 1; i <= mitad; i++, j++ )
68          System.out.println( lista.get( i ).toString() + lista.get( j ) );
```

Figura 22.9 Ejemplo para barajar y repartir cartas mediante el método **shuffle** de **Collections**. (Parte 2 de 3.)

```
69     }
70
71     public static void main( String args[] )
72     {
73        new Cartas().imprimirCartas();
74     }
75
76  } // fin de la clase Cartas
```

```
Cuatro de Espadas    As de Espadas
Rey de Diamantes     Siete de Diamantes
Diez de Corazones    Joto de Espadas
Ocho de Corazones    Siete de Treboles
Tres de Espadas      Cinco de Espadas
Diez de Espadas      Joto de Diamantes
Dos de Corazones     Quina de Corazones
Cuatro de Corazones  Nueve de Treboles
Cinco de Corazones   Siete de Espadas
Seis de Corazones    Nueve de Diamantes
Cinco de Diamantes   Quina de Espadas
Ocho de Diamantes    Dos de Diamantes
Rey de Treboles      Cuatro de Treboles
Quina de Treboles    Seis de Espadas
Joto de Treboles     Siete de Corazones
Rey de Espadas       Quina de Diamantes
As de Corazones      Seis de Treboles
Rey de Corazones     Nueve de Espadas
Joto de Corazones    Cinco de Treboles
Ocho de Espadas      Dos de Espadas
Diez de Diamantes    As de Treboles
Nueve de Corazones   Ocho de Treboles
Tres de Treboles     Dos de Treboles
Cuatro de Diamantes  Tres de Diamantes
Diez de Treboles     Tres de Corazones
Seis de Diamantes    As de Diamantes
```

Figura 22.9 Ejemplo para barajar y repartir cartas mediante el método **shuffle** de **Collections**. (Parte 3 de 3.)

22.6.3 Los algoritmos reverse, fill, copy, max y min

La clase **Collections** proporciona algoritmos para invertir, llenar y copiar objetos **List**. El algoritmo **reverse** invierte el orden de los elementos en un objeto **List** y el algoritmo **fill** sobrescribe los elementos en un objeto **List** con un valor especificado. (La operación **fill** es útil para reinicializar un objeto **List**.) El algoritmo **copy** toma dos argumentos: un objeto **List** de destino y un objeto **List** de origen. Cada elemento del objeto **List** de origen se copia al objeto **List** de destino. El objeto **List** de destino debe tener cuando menos la misma longitud que el objeto **List** de origen; de lo contrario, se producirá una excepción **IndexOutOfBoundsException**. Si el objeto **List** de destino es más largo, los elementos que no se sobrescriban permanecerán sin cambio.

Cada uno de los algoritmos que hemos visto hasta ahora operan en objetos **List**. Los algoritmos **min** y **max** operan en cualquier objeto **Collection**. El algoritmo **min** devuelve el elemento más pequeño de un objeto **List** (recuerde que un objeto **List** es un objeto **Collection**), y el algoritmo **max** devuelve el elemento más grande en un objeto **List**. Ambos algoritmos pueden llamarse con un objeto **Comparator** como segundo argumento, para realizar comparaciones personalizadas entre objetos. En la figura 22.10 se demuestra el uso de los algoritmos **reverse**, **fill**, **copy**, **min** y **max**.

http://librosysolucionarios.net

En la línea 19 se hace una llamada al método ***reverse*** de **Collections** para invertir el orden de **lista**. El método **reverse** toma un argumento **List**. (En este caso, **lista** es una vista tipo **List** del arreglo **letras**.) El arreglo **letras** ahora tiene sus elementos en orden inverso.

En la línea 23 se copian los elementos de **lista** a **listaCopia**, utilizando el método **copy** de **Collections**. Los cambios a **listaCopia** no provocan cambios en **letras**; **listaCopia** es un objeto **List** separado que no es una vista tipo **List** para **letras**. El método **copy** requiere dos argumentos **List**.

En la línea 27 se hace una llamada al método **fill** de **Collections** para colocar la cadena **"R"** en cada elemento de **lista**. Como **lista** es una vista tipo **List** de **letras**, esta operación cambia cada elemento en **letras** a **"R"**. El método **fill** requiere un objeto **List** como primer argumento, y un objeto **Object** como segundo argumento.

En las líneas 41 y 42 se hace una llamada a los métodos **max** y **min** de **Collection** para encontrar el elemento más grande y más pequeño, respectivamente.

```
1  // Fig. 22.10: Algoritmos1.java
2  // Uso de los algoritmos reverse, fill, copy, min y max.
3  import java.util.*;
4
5  public class Algoritmos1 {
6     private String letras[] = { "P", "C", "M" }, copiaLetras[];
7     private List lista, listaCopia;
8
9     // crear un objeto List y manipularlo con métodos de las colecciones
10    public Algoritmos1()
11    {
12       lista = Arrays.asList( letras );      // obtener objeto List
13       copiaLetras = new String[ 3 ];
14       listaCopia = Arrays.asList( copiaLetras );
15
16       System.out.println( "Lista inicial: " );
17       mostrar( lista );
18
19       Collections.reverse( lista );         // invertir el orden
20       System.out.println( "\nDespues de llamar a reverse: " );
21       mostrar( lista );
22
23       Collections.copy( listaCopia, lista );   // copiar objeto List
24       System.out.println( "\nDespues de copiar: " );
25       mostrar( listaCopia );
26
27       Collections.fill( lista, "R" );          // llenar lista con Rs
28       System.out.println( "\nDespues de llamar a fill: " );
29       mostrar( lista );
30
31    } // fin del constructor
32
33    // mostrar información del objeto List
34    private void mostrar( List refLista )
35    {
36       System.out.print( "La lista es: " );
37
```

Figura 22.10 Los métodos **reverse**, **fill**, **copy**, **max** y **min** de **Collections**. (Parte 1 de 2.)

```
38          for ( int k = 0; k < refLista.size(); k++ )
39             System.out.print( refLista.get( k ) + " " );
40
41          System.out.print( "\nMax: " + Collections.max( refLista ) );
42          System.out.println( "  Min: " + Collections.min( refLista ) );
43       }
44
45       public static void main( String args[] )
46       {
47          new Algoritmos1();
48       }
49
50    } // fin de la clase Algoritmos1
```

```
Lista inicial:
La lista es: P C M
Max: P  Min: C

Despues de llamar a reverse:
La lista es: M C P
Max: P  Min: C

Despues de copiar:
La lista es: M C P
Max: P  Min: C

Despues de llamar a fill:
La lista es: R R R
Max: R  Min: R
```

Figura 22.10 Los métodos **reverse**, **fill**, **copy**, **max** y **min** de **Collections**. (Parte 2 de 2.)

22.6.4 El algoritmo `binarySearch`

En la sección 7.8 estudiamos el algoritmo de búsqueda binaria, de alta velocidad. Este algoritmo se incluye en el marco de trabajo de colecciones de Java como un método estático de la clase **Collections**. El algoritmo **binarySearch** localiza un objeto **Object** en un objeto **List** (es decir, un objeto **LinkedList**, **Vector** o **ArrayList**). Si se encuentra el objeto **Object**, se devuelve el índice de ese objeto. Si no se encuentra el objeto **Object**, **binarySearch** devuelve un valor negativo. El algoritmo **binarySearch** determina este valor negativo calculando primero el punto de inserción y cambiando el signo del punto de inserción a negativo. Por último, **binarySearch** resta uno al punto de inserción para obtener el valor de retorno. Si varios elementos en la lista concuerdan con la clave de búsqueda, no hay garantía de que uno se localice primero. En la figura 22.11 se utiliza el algoritmo **binarySearch** para buscar una serie de cadenas en un objeto **ArrayList**.

Antes de realizar búsquedas con el método de búsqueda binaria, los elementos de la lista deben estar ordenados en forma ascendente, por lo que en la línea 14 el constructor ordena la lista mediante el método **sort** de **Collections**. En la línea 15 se muestra la lista ordenada. El método **imprimirResultadosBusqueda** (líneas 19 a 27) es llamado desde **main** para realizar las búsquedas. Cada búsqueda hace una llamada al método **ayudanteImprimirResultadosBusqueda** (líneas 30 a 38) para realizar la búsqueda y mostrar los resultados. En la línea 35 se hace una llamada al método **binarySearch** de **Collections** para buscar en **lista** la **clave** especificada. El método **binarySearch** toma un objeto **List** como primer argumento y un objeto **Object** como segundo argumento. En las líneas 36 y 37 se muestran los resultados de la búsqueda. Una versión sobrecargada de **binarySearch** toma un objeto **Comparator** como tercer argumento, el cual especifica la manera en que **binarySearch** debe comparar los elementos.


```
1  // Fig. 22.11: PruebaBinarySearch.java
2  // Uso del algoritmo binarySearch.
3  import java.util.*;
4
5  public class PruebaBinarySearch {
6     private static final String colores[] = { "rojo", "blanco",
7        "azul", "negro", "amarillo", "morado", "carne", "rosa" };
8     private List lista;         // referencia ArrayList
9
10    // crear, ordenar y mostrar lista
11    public PruebaBinarySearch()
12    {
13       lista = new ArrayList( Arrays.asList( colores ) );
14       Collections.sort( lista ); // ordenar el objeto ArrayList
15       System.out.println( "Objeto ArrayList ordenado: " + lista );
16    }
17
18    // buscar en lista varios valores
19    private void imprimirResultadosBusqueda()
20    {
21       ayudanteImprimirResultadosBusqueda( colores[ 3 ] ); // primer elemento
22       ayudanteImprimirResultadosBusqueda( colores[ 0 ] ); // elemento de en medio
23       ayudanteImprimirResultadosBusqueda( colores[ 7 ] ); // último elemento
24       ayudanteImprimirResultadosBusqueda( "hormiguero" ); // más abajo del menor
25       ayudanteImprimirResultadosBusqueda( "cabra" );      // no existe
26       ayudanteImprimirResultadosBusqueda( "cebra" );      // no existe
27    }
28
29    // método ayudante para realizar búsquedas
30    private void ayudanteImprimirResultadosBusqueda( String clave )
31    {
32       int resultado = 0;
33
34       System.out.println( "\nBuscando: " + clave );
35       resultado = Collections.binarySearch( lista, clave );
36       System.out.println( ( resultado >= 0 ? "Se encontro en el indice " + resultado :
37          "No se encontro (" + resultado + ")" ) );
38    }
39
40    public static void main( String args[] )
41    {
42       new PruebaBinarySearch().imprimirResultadosBusqueda();
43    }
44
45 } // fin de la clase PruebaBinarySearch
```

```
Objeto ArrayList ordenado: [amarillo, azul, blanco, carne, morado, negro, rojo,
rosa]

Buscando: negro
Se encontro en el indice 5                    (Continúa en la siguiente página)
```

Figura 22.11 El método **binarySearch** de **Collections**. (Parte 1 de 2.)

http://librosysolucionarios.net

```
Buscando: rojo
Se encontro en el indice 6

Buscando: rosa
Se encontro en el indice 7

Buscando: hormiguero
No se encontro (-5)

Buscando: cabra
No se encontro (-4)

Buscando: cebra
No se encontro (-5)
```

Figura 22.11 El método **binarySearch** de **Collections**. (Parte 2 de 2.)

22.7 Conjuntos

Un objeto ***Set*** es un objeto **Collection** que contiene elementos únicos (es decir, sin elementos duplicados). El marco de trabajo de colecciones contiene varias implementaciones de **Set**, incluyendo a ***HashSet*** y ***TreeSet***. **HashSet** almacena sus elementos en una tabla de hash, y **TreeSet** almacena sus elementos en un árbol. En la figura 22.12 se utiliza un objeto **HashSet** para eliminar las cadenas duplicadas de un objeto **ArrayList**.

```
1  // Fig. 22.12: PruebaSet.java
2  // Uso de un objeto HashSet para eliminar duplicados.
3  import java.util.*;
4
5  public class PruebaSet {
6     private static final String colores[] = { "rojo", "blanco", "azul",
7        "verde", "gris", "anaranjado", "carne", "blanco", "cyan",
8        "durazno", "gris", "anaranjado" };
9
10    // crear y mostrar objeto ArrayList
11    public PruebaSet()
12    {
13       List lista = new ArrayList( Arrays.asList( colores ) );
14       System.out.println( "ArrayList: " + lista );
15       imprimirSinDuplicados( lista );
16    }
17
18    // crear conjunto a partir del arreglo para eliminar dup.
19    private void imprimirSinDuplicados( Collection coleccion )
20    {
21       // crear un objeto HashSet y obtener su iterador
22       Set conjunto = new HashSet( coleccion );
23       Iterator iterador = conjunto.iterator();
24
25       System.out.println( "\nLos no duplicados son: " );
26
```

Figura 22.12 Objeto **HashSet** utilizado para eliminar valores duplicados de un arreglo de cadenas. (Parte 1 de 2.)

http://librosysolucionarios.net


```
27       while ( iterador.hasNext() )
28          System.out.print( iterador.next() + " " );
29
30       System.out.println();
31    }
32
33    public static void main( String args[] )
34    {
35       new PruebaSet();
36    }
37
38 } // fin de la clase PruebaSet
```

```
ArrayList: [rojo, blanco, azul, verde, gris, anaranjado, carne, blanco, cyan,
durazno, gris, anaranjado]

Los no duplicados son:
gris rojo cyan carne azul anaranjado verde blanco durazno
```

Figura 22.12 Objeto **HashSet** utilizado para eliminar valores duplicados de un arreglo de cadenas. (Parte 2 de 2.)

El método **imprimirSinDuplicados** (líneas 19 a 31), el cual es llamado desde el constructor, toma un argumento **Collection**. En la línea 22 se crea un objeto **HashSet** a partir del argumento **Collection**. Por definición, los objetos **Set** no contienen valores duplicados. Por lo tanto, cuando se construye el objeto **HashSet**, éste elimina cualquier valor duplicado en el objeto **Collection**. En la línea 23 se obtiene un objeto **Iterator** para el objeto **Set**. El ciclo en las líneas 27 y 28 llama a los métodos **hasNext** y **next** de **Iterator** para acceder a los elementos del objeto **Set** y mostrar sus valores.

Conjuntos ordenados

El marco de trabajo de colecciones también incluye la interfaz ***SortedSet*** (que extiende a **Set**) para los conjuntos que mantengan a sus elementos ordenados; ya sea en el orden natural de los elementos (por ejemplo, los números se encuentran en orden ascendente) o en un orden especificado por un objeto **Comparator**. La clase **TreeSet** implementa a **SortedSet**. El programa de la figura 22.13 coloca cadenas en un objeto **TreeSet**. Estas cadenas se ordenan al ser agregadas al objeto **TreeSet**. Este ejemplo también demuestra los *métodos de vista de rango*, los cuales permiten a un programa ver una porción de una colección.

```
1  // Fig. 22.13: PruebaSortedSet.java
2  // Uso de TreeSet y SortedSet.
3  import java.util.*;
4
5  public class PruebaSortedSet {
6     private static final String nombres[] = { "amarillo", "verde",
7          "negro", "carne", "gris", "blanco", "anaranjado", "rojo", "verde" };
8
9     // crear un conjunto ordenado con TreeSet, después manipularlo
10    public PruebaSortedSet()
11    {
12       SortedSet arbol = new TreeSet( Arrays.asList( nombres ) );
13
```

Figura 22.13 Uso de objetos **SortedSet** y **TreeSet**. (Parte 1 de 2.)

```
14          System.out.println( "conjunto: " );
15          imprimirConjunto( arbol );
16
17          // obtener conjunto headSet con base en "carne"
18          System.out.print( "\nheadSet (\"carne\"):  " );
19          imprimirConjunto( arbol.headSet( "carne" ) );
20
21          // obtener conjunto tailSet con base en "carne"
22          System.out.print( "tailSet (\"carne\"):  " );
23          imprimirConjunto( arbol.tailSet( "carne" ) );
24
25          // obtener elementos primero y último
26          System.out.println( "primero: " + arbol.first() );
27          System.out.println( "ultimo : " + arbol.last() );
28       }
29
30       // mostrar conjunto
31       private void imprimirConjunto( SortedSet conjunto )
32       {
33          Iterator iterador = conjunto.iterator();
34
35          while ( iterador.hasNext() )
36             System.out.print( iterador.next() + " " );
37
38          System.out.println();
39       }
40
41       public static void main( String args[] )
42       {
43          new PruebaSortedSet();
44       }
45
46    } // fin de la clase PruebaSortedSet
```

```
conjunto:
amarillo anaranjado blanco carne gris negro rojo verde

headSet ("carne"):  amarillo anaranjado blanco
tailSet ("carne"):  carne gris negro rojo verde
primero: amarillo
ultimo : verde
```

Figura 22.13 Uso de objetos `SortedSet` y `TreeSet`. (Parte 2 de 2.)

En la línea 12 del constructor se crea un objeto `TreeSet` que contiene los elementos del arreglo `nombres` y se asigna el objeto `SortedSet` a la referencia `arbol`. En la línea 15 se muestra el conjunto inicial de cadenas utilizando el método `imprimirConjunto` (líneas 31 a 39), sobre el cual hablaremos en breve. En la línea 19 se hace una llamada al método *`headSet`* de `TreeSet` para obtener un subconjunto del objeto `TreeSet`, en el que todos los elementos serán menores que `"carne"`. La vista devuelta de `headSet` se muestra a continuación con `imprimirConjunto`. Si se hace algún cambio al subconjunto, éste se reflejará también en el objeto `TreeSet` original. Por lo tanto, el subconjunto devuelto por `headSet` incluirá los objeto `TreeSet`.

http://libros.sysolucionarios.net

En la línea 23 se hace una llamada al método *`tailSet`* de `TreeSet` para obtener un subconjunto en el que cada elemento sea mayor o igual que `"carne"`. Después, el subconjunto devuelto por `tailSet` se muestra en pantalla. Cualquier cambio realizado a través de la vista `tailSet` se realiza también en el objeto `TreeSet` original. En las líneas 26 y 27 se hace una llamada a los métodos *`first`* y *`last`* de `SortedSet` para obtener el elemento más pequeño y más grande del conjunto, respectivamente.

El método `imprimirConjunto` (líneas 31 a 39) recibe un objeto `SortedSet` como argumento y lo imprime. En la línea 33 se obtiene un objeto `Iterator` para el objeto `Set`, el cual se utiliza en las líneas 35 y 36 para imprimir cada elemento del objeto `SortedSet`.

22.8 Mapas

Los objetos *`Map`* asocian claves a valores y no pueden contener claves duplicadas (es decir, cada clave puede asociarse solamente con un valor; a este tipo de asociación se le conoce como *asociación de uno a uno*.) Los objetos `Map` difieren de los objetos `Set` en cuanto a que los primeros contienen claves y valores, mientras que los segundos contienen solamente valores. Dos de las muchas clases que implementan a la interfaz `Map` son *`HashMap`* y *`TreeMap`*. Los objetos `HashMap` almacenan elementos en tablas de hash, y los objetos `TreeMap` almacenan elementos en árboles. La interfaz *`SortedMap`* extiende a `Map` y mantiene sus claves en orden; ya sea el orden natural de los elementos o un orden especificado por un objeto `Comparator`. La clase `TreeMap` implementa a `SortedMap`. En la figura 22.14 se utiliza un objeto `HashMap` para contar el número de ocurrencias de cada palabra en una cadena. La clase `HashMap` permite el uso de una clave `null` y de valores `null`.

```
1  // Fig. 22.14: CuentaTipoPalabras.java
2  // Programa que cuenta el número de ocurrencias de cada palabra en una cadena
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7
8  public class CuentaTipoPalabras extends JFrame {
9     private JTextArea campoEntrada;
10    private JLabel indicador;
11    private JTextArea pantalla;
12    private JButton botonIniciar;
13
14    private Map mapa;
15
16    public CuentaTipoPalabras()
17    {
18       super( "Cuenta del tipo de palabras" );
19       campoEntrada = new JTextArea( 3, 20 );
20
21       mapa = new HashMap();
22
23       botonIniciar = new JButton( "Iniciar" );
24       botonIniciar.addActionListener(
25
26          new ActionListener() { // clase interna
27
28             public void actionPerformed( ActionEvent evento )
29             {
30                crearMapa();
31                pantalla.setText( crearSalida() );
```

Figura 22.14 Objetos `HashMap` y `Map`. (Parte 1 de 3.)

```
32              }
33
34           } // fin de la clase interna
35
36        }; // fin de la llamada a addActionListener
37
38        indicador = new JLabel( "Escriba una cadena:" );
39        pantalla = new JTextArea( 15, 20 );
40        pantalla.setEditable( false );
41
42        JScrollPane mostrarPanelDesplazable = new JScrollPane( pantalla );
43
44        // agregar componentes a GUI
45        Container contenedor = getContentPane();
46        contenedor.setLayout( new FlowLayout() );
47        contenedor.add( indicador );
48        contenedor.add( campoEntrada );
49        contenedor.add( botonIniciar );
50        contenedor.add( mostrarPanelDesplazable );
51
52        setSize( 450, 400 );
53        show();
54
55     } // fin del constructor
56
57     // crear mapa a partir de la entrada del usuario
58     private void crearMapa()
59     {
60        String entrada = campoEntrada.getText();
61        StringTokenizer tokenizer = new StringTokenizer( entrada );
62
63        while ( tokenizer.hasMoreTokens() ) {
64           String palabra = tokenizer.nextToken().toLowerCase(); // obtener palabra
65
66           // si el mapa contiene la palabra
67           if ( mapa.containsKey( palabra ) ) {
68
69              Integer cuenta = (Integer) mapa.get( palabra ); // obtener valor
70
71              // incrementar valor
72              mapa.put( palabra, new Integer( cuenta.intValue() + 1 ) );
73           }
74           else // en caso contrario agregar palabra con valor de 1 a mapa
75              mapa.put( palabra, new Integer( 1 ) );
76
77        } // fin de instrucción while
78
79     } // fin del método crearMapa
80
81     // crear cadena que contenga valores de mapa
82     private String crearSalida() {
83        StringBuffer salida = new StringBuffer( "" );
84        Iterator claves = mapa.keySet().iterator();
85
86        // iterar a través de las claves
```

Figura 22.14 Objetos **HashMap** y **Map**. (Parte 2 de 3.)


```
87        while ( claves.hasNext() ) {
88           Object claveActual = claves.next();
89
90           // mostrar los pares clave-valor
91           salida.append( claveActual + "\t" +
92                          mapa.get( claveActual ) + "\n" );
93        }
94
95        salida.append( "size: " + mapa.size() + "\n" );
96        salida.append( "isEmpty: " + mapa.isEmpty() + "\n" );
97
98        return salida.toString();
99
100    } // fin del método crearSalida
101
102    public static void main( String args[] )
103    {
104       CuentaTipoPalabras aplicacion = new CuentaTipoPalabras();
105       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
106    }
107
108 } // fin de la clase CuentaTipoPalabras
```

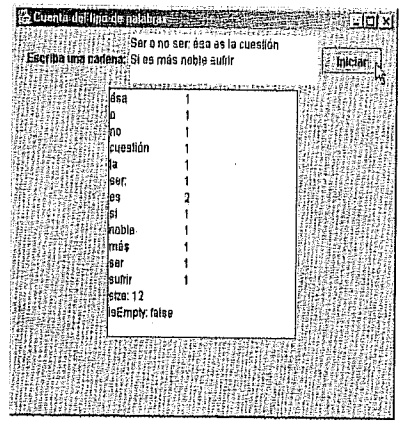


Figura 22.14 Objetos **HashMap** y **Map**. (Parte 3 de 3.)

En la línea 21 se crea un objeto **HashMap** y se asigna a **mapa**. El usuario escribe una oración en el área de texto que se encuentra en la parte superior de la ventana y después oprime el botón **Iniciar** para invocar a **actionPerformed** (líneas 28 a 32). En la línea 30 se hace una llamada al método **crearMapa** (líneas 58 a 79), el cual utiliza un **mapa** para almacenar el número de ocurrencias de cada palabra en la oración. En la línea 60 se obtiene la entrada del usuario y en la línea 61 se crea un objeto **StringTokenizer** para dividir la cadena en tokens o palabras. Mientras haya más tokens, la línea 64 convierte el siguiente token en minúsculas. Después, en la línea 67 se hace una llamada al método ***containsKey*** de **Map** para determinar si la palabra se encuentra en el mapa (y por lo tanto ha ocurrido anteriormente en la cadena). Si el objeto **Map** no contiene una asociación para la palabra, en la línea 75 se utiliza el método ***put*** de **Map** para crear una nueva entrada en el mapa, con la palabra como la clave y un objeto **Integer** que contiene **1** como el valor. Si la palabra ya existe en el mapa, en la línea 69 se utiliza el método **get** de **Map** para obtener el valor asociado a la clave en el mapa. En la línea 72 se incrementa ese valor y se utiliza **put** para reemplazar el valor asociado de la clave

http://librosysolucionarios.net

en el mapa. El método `put` devuelve el valor anterior asociado con la clave, o `null` si la clave no se encuentra en el mapa.

El método `crearSalida` (líneas 82 a 100) crea una cadena con todas las entradas en el mapa. Utiliza el método *`keySet`* de `HashMap` (línea 84) para obtener un conjunto de las claves. Después utiliza un iterador para ese conjunto en un ciclo (líneas 87 a 93) para acceder a cada clave en el mapa. En la línea 91 se anexa cada clave con su valor asociado a la cadena `salida`. En las líneas 95 y 96 se hace una llamada a los métodos *`size`* e *`isEmpty`* de `Map` para obtener el número de pares clave-valor en el objeto `Map` y un valor `boolean` (que indica si el objeto `Map` está vacío) respectivamente.

22.9 Envolturas de sincronización

En el capítulo 16 hablamos sobre el subprocesamiento múltiple. Las colecciones en el marco de trabajo de colecciones están desincronizadas de manera predeterminada, por lo que pueden operar eficientemente. Sin embargo, como están desincronizadas el acceso concurrente a un objeto `Collection` por parte de varios subprocesos podría ocasionar resultados indeterminados o errores fatales. Para evitar potenciales problemas con subprocesos, se utilizan envolturas de sincronización para las colecciones que podrían ser utilizadas por varios subprocesos. Una *clase de envoltura* recibe las llamadas a métodos, agrega la sincronización de subprocesos para que éstos operen en forma segura y delega las llamadas a la clase de envoltura. La API `Collections` proporciona un conjunto de métodos estáticos para convertir las colecciones a sus versiones sincronizadas. Los encabezados de método para las envolturas de sincronización se enlistan en la figura 22.15.

Encabezado de método `public static`
`Collection synchronizedCollection( Collection c )`
`List synchronizedList( List unaLista )`
`Set synchronizedSet( Set s )`
`SortedSet synchronizedSortedSet( SortedSet s )`
`Map synchronizedMap( Map m )`
`SortedMap synchronizedSortedMap( SortedMap m )`

Figura 22.15 Métodos de envolturas de sincronización.

22.10 Envolturas no modificables

La API `Collections` proporciona un conjunto de métodos estáticos para convertir las colecciones a versiones no modificables (llamadas *envolturas no modificables*) de esas colecciones. Las envolturas no modificables lanzan excepciones `UnsupportedOperationException` si se trata de modificar la colección. Los encabezados de estos métodos se enlistan en la figura 22.16.

Encabezado de método `public static`
`Collection unmodifiableCollection( Collection c )`
`List unmodifiableList( List unaLista )`
`Set unmodifiableSet( Set s )`
`SortedSet unmodifiableSortedSet( SortedSet s )`
`Map unmodifiableMap( Map m )`
`SortedMap unmodifiableSortedMap( SortedMap m )`

Figura 22.16 Métodos de envolturas no modificables.

http://librosysolucionarios.net



Observación de ingeniería de software 22.7

Usted puede utilizar una envoltura no modificable para crear una colección que ofrezca acceso de sólo lectura a otros y le permita a usted el acceso de lectura-escritura. Para ello, simplemente necesita proporcionar a los otros una referencia a la envoltura no modificable y retener para usted mismo una referencia a la colección envuelta.

22.11 Implementaciones abstractas

El marco de trabajo de colecciones proporciona varias implementaciones abstractas de interfaces de colecciones, a partir de las cuales el programador puede obtener rápidamente implementaciones personalizadas completas. Estas implementaciones abstractas son: una implementación delgada de `Collection` conocida como *`AbstractCollection`*; una implementación delgada de `List` con respaldo de acceso aleatorio, conocida como *`AbstractList`*; una implementación delgada de `Map` conocida como *`AbstractMap`*, una implementación delgada de `List` con respaldo de acceso secuencial, conocida como *`AbstractSequentialList`*; y una implementación delgada de `Set` conocida como *`AbstractSet`*.

Para escribir una implementación personalizada, empiece extendiendo la clase de implementación abstracta que se adapte mejor a sus necesidades. Después implemente cada uno de los métodos `abstract` de esa clase. A continuación, si su colección va a ser modificable, redefina cualquier método concreto que evite la modificación.

22.12 (Opcional) Descubrimiento de los patrones de diseño: Patrones de diseño utilizados en el paquete `java.util`

En esta sección utilizaremos el material acerca de las estructuras de datos y colecciones descrito en los capítulos 20 a 22 para identificar las clases del paquete `java.util` que utilizan patrones de diseño. Esta sección concluye nuestro análisis acerca de los patrones de diseño.

22.12.1 Patrones de diseño de creación

Concluiremos nuestra discusión sobre los patrones de diseño de creación hablando sobre el patrón de diseño Prototipo.

Prototipo

Algunas veces un sistema debe realizar una copia de un objeto, pero no "conocerá" la clase de ese objeto sino hasta en tiempo de ejecución. Por ejemplo, considere el diseño del programa de dibujo del ejercicio 10.9; las clases `MiLinea`, `MiOvalo` y `MiRect` representan clases de "figuras" que extienden a la superclase abstracta `MiFigura`. Podríamos modificar este ejercicio para permitir al usuario crear, copiar y pegar nuevas instancias de la clase `MiLinea` en el programa. El *patrón de diseño Prototipo* permite que un objeto (llamado *prototipo*) devuelva una copia de ese prototipo a un objeto que lo solicite (llamado *cliente*). Cada prototipo debe pertenecer a una clase que implemente una interfaz común, la cual debe permitir al prototipo clonarse a sí mismo. Por ejemplo, la API de Java proporciona el método `clone` de la clase `java.lang.Object` y la interfaz `java.lang.Cloneable` (cualquier objeto de una clase que implemente a `Cloneable` puede utilizar el método `clone` para copiarse a sí mismo). Específicamente, el método `clone` crea una copia de un objeto y después devuelve una referencia a ese objeto. Si diseñamos la clase `MiLinea` como el prototipo para el ejercicio 10.9, entonces esta clase debe implementar a la interfaz `Cloneable`. Para crear una nueva línea en nuestro dibujo, clonamos el prototipo `MiLinea`. Para copiar una línea ya existente, clonamos ese objeto. El método `clone` también es útil en métodos que devuelven una referencia a un objeto, cuando el desarrollador no desea que ese objeto se altere a través de esa referencia; el método `clone` devuelve una referencia a la copia del objeto, en vez de devolver la referencia a ese objeto. Para obtener más información acerca de la interfaz `Cloneable`, visite la página:

`www.java.sun.com/j2se/1.4/docs/api/java/lang/Cloneable.html`

22.12.2 Patrones de diseño de comportamiento

Concluiremos nuestra discusión sobre los patrones de diseño de comportamiento hablando sobre el patrón de diseño Iterador.

Iterador

Los diseñadores utilizan estructuras de datos como arreglos, listas enlazadas y tablas de hash para organizar los datos en un programa. El *patrón de diseño Iterador* permite a los objetos acceder a objetos individuales de cualquier estructura de datos sin tener que "conocer" el comportamiento de esa estructura de datos (como recorrer la estructura o eliminar un elemento de esa estructura) o la manera en que ésta almacena los objetos. Las instrucciones para recorrer la estructura de datos y acceder a sus elementos se almacenan en un objeto separado, conocido como *iterador*. Cada estructura de datos puede crear un iterador; cada iterador implementa métodos de una interfaz común para recorrer la estructura de datos y acceder a sus datos. Un cliente puede recorrer dos estructuras de datos con distinta estructura (tal como una lista enlazada y una tabla de hash) en la misma forma, ya que ambas estructuras de datos proporcionan un objeto iterador que pertenece a una clase que implementa a una interfaz común. Java proporciona la interfaz `Iterator` del paquete `java.util`, la cual vimos en la sección 22.5; la clase `PruebaCollection` (figura 22.3) utiliza un objeto `Iterator`.

22.12.3 Conclusión

En nuestras secciones opcionales "Descubrimiento de los patrones de diseño" hemos presentado la importancia, utilidad y predominio de los patrones de diseño. Hemos mencionado que en su libro titulado *Design Patterns, Elements of Reusable Object-Oriented Software*, la "Banda de los cuatro" describió 23 patrones de diseño que ofrecen estrategias comprobadas para crear sistemas. Cada patrón pertenece a una de tres categorías de patrones: de creación, que trata las cuestiones relacionadas con la creación de objetos; de estructura, que proporciona maneras de organizar clases y objetos en un sistema; y de comportamiento, que ofrece estrategias para modelar la manera en que los objetos colaboran entre sí en un sistema.

De los 23 patrones de diseño, vimos 18 de los más populares que son utilizados por la comunidad de Java. En las secciones 10.12, 14.14, 16.12, 18.12 y 22.12 dividimos la discusión de acuerdo con la manera en que ciertos paquetes de Java (como los paquetes `java.awt`, `javax.swing`, `java.io`, `java.net` y `java.util`) utilizan estos patrones de diseño. También hablamos sobre los patrones que la "Banda de los cuatro" no describió, como lo patrones de concurrencia, que son útiles en sistemas con subprocesamiento múltiple; y los patrones arquitectónicos, que ayudan a los diseñadores a asignar cierta funcionalidad a varios subsistemas dentro de un sistema. Hemos motivado cada patrón (es decir, hemos explicado por qué ese patrón es importante y cómo puede utilizarse). Hemos proporcionado, según sea apropiado, varios ejemplos en forma de analogías del mundo real (por ejemplo, el adaptador en el patrón de diseño Adaptador es similar a un adaptador para un enchufe en un dispositivo eléctrico). También proporcionamos ejemplos acerca de cómo los paquetes de Java aprovechan el uso de los patrones de diseño (por ejemplo, los componentes de la GUI de Swing utilizan el patrón de diseño Observador para colaborar con sus componentes de escucha para responder a las interacciones de los usuarios). Además proporcionamos ejemplos sobre cómo ciertos programas en *Cómo programar en Java, quinta edición* utilizan los patrones de diseño (por ejemplo, el ejemplo práctico del elevador en nuestras secciones opcionales "Acerca de los objetos" utiliza el patrón de diseño Estado para representar la ubicación de un objeto `Persona` en la simulación).

Esperamos que vea nuestras secciones "Descubrimiento de los patrones de diseño" como el comienzo de un estudio detallado sobre los patrones de diseño. Si no lo ha hecho ya, le recomendamos que visite las diversas URLs que hemos proporcionado en la sección 10.12.5, Recursos en Internet y World Wide Web. Le recomendamos que después lea el libro de la Banda de los cuatro. Esta información le ayudará a crear mejores sistemas, utilizando la sabiduría colectiva de la industria de la tecnología de objetos.

Si estudió las secciones opcionales de este libro, entonces ha obtenido una introducción a sistemas de Java más substanciales. Si leyó nuestras secciones opcionales "Acerca de los objetos", entonces se ha infiltrado en una verdadera experiencia de diseño e implementación en Java, habiendo aprendido una metodología disciplinada para el diseño orientado a objetos con UML. Si leyó nuestras secciones opcionales "Descubrimiento de los patrones de diseño", entonces ha elevado su conocimiento acerca del tema más avanzado de los patrones de diseño.

Esperamos que continúe con su estudio sobre los patrones de diseño, y le agradeceríamos enormemente si enviara sus comentarios, críticas y sugerencias para mejorar *Cómo programar en Java* a deitel@deitel.com. ¡Buena suerte!

http://libros.sysolucionarios.net

RESUMEN

- El marco de trabajo de colecciones de Java proporciona al programador acceso a estructuras de datos preempaquetadas y algoritmos para manipular a esas estructuras de datos.
- Una colección es un objeto que puede guardar referencias a otros objetos. Las interfaces de las colecciones declaran las operaciones que pueden realizarse en cada tipo de colección.
- Las clases e interfaces del marco de trabajo de colecciones se encuentran en el paquete `java.util`.
- La clase `Arrays` proporciona métodos estáticos para manipular arreglos, incluyendo a `sort` para ordenar un arreglo, `binarySearch` para buscar en un arreglo ordenado, `equals` para comparar arreglos y `fill` para colocar elementos en un arreglo.
- El método `asList` de `Arrays` devuelve una vista `List` de un arreglo, la cual permite a un programa manipular el arreglo como si fuera un objeto `List`. Cualquier modificación realizada a través de la vista `List` cambia al arreglo, y cualquier modificación realizada al arreglo cambia a la vista `List`.
- El método `size` obtiene el número de elementos en un objeto `List`, y el método `get` devuelve un elemento `List`.
- La interfaz `Collection` es la interfaz raíz en la jerarquía de colecciones, a partir de la cual se derivan las interfaces `Set` y `List`. La interfaz `Collection` contiene operaciones masivas para agregar, borrar, comparar y retener objetos en una colección. La interfaz `Collection` proporciona un método llamado `iterator` para obtener un objeto `Iterator`.
- La clase `Collections` proporciona métodos estáticos para manipular colecciones. Muchos de los métodos son implementaciones de algoritmos polimórficos para buscar, ordenar, etcétera.
- Un objeto `List` es un objeto `Collection` ordenado que puede contener elementos duplicados.
- La interfaz `List` es implementada por las clases `ArrayList`, `LinkedList` y `Vector`. La clase `ArrayList` es una implementación de un objeto `List` como arreglo que puede modificar su tamaño. Un objeto `LinkedList` es una implementación de un objeto `List` como lista enlazada.
- El método `hasNext` de `Iterator` determina si un objeto `Collection` contiene otro elemento. El método `next` devuelve una referencia al siguiente objeto en la colección del objeto `Collection`, y avanza el objeto `Iterator`.
- El método `subList` devuelve una vista de parte de un objeto `List`. Cualquier cambio realizado a una sublista se hace también en el objeto `List`.
- El método `clear` elimina elementos de un objeto `List`.
- El método `toArray` devuelve el contenido de una colección como un arreglo.
- Los algoritmos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` y `copy` operan en objetos `List`. Los algoritmos `min` y `max` operan en objetos `Collection`. El algoritmo `reverse` invierte los elementos de un objeto `List`; `fill` asigna cada elemento de un objeto `List` a un objeto `Object` especificado; y `copy` copia elementos de un objeto `List` a otro objeto `List`. El algoritmo `sort` ordena los elementos de un objeto `List`.
- Los algoritmos `min` y `max` encuentran el elemento más pequeño y el más grande en un objeto `Collection`.
- El objeto `Comparator` proporciona un medio para ordenar los elementos de un objeto `Collection` en un orden distinto al orden natural de los elementos.
- El método `reverseOrder` de `Collections` devuelve un objeto `Comparator` que puede utilizarse con `sort` para ordenar los elementos de una colección en forma inversa.
- El algoritmo `shuffle` ordena al azar los elementos de un objeto `List`.
- El algoritmo `binarySearch` localiza un objeto `Object` en un objeto `List` ordenado.
- Un objeto `Set` es un objeto `Collection` que no contiene elementos duplicados. `HashSet` almacena sus elementos en una tabla de hash. `TreeSet` almacena sus elementos en un árbol.
- La interfaz `SortedSet` extiende a `Set` y representa un conjunto que mantiene sus elementos ordenados. La clase `TreeSet` implementa a `SortedSet`.
- El método `headSet` de `TreeSet` obtiene un subconjunto de un objeto `TreeSet` en el que todos sus elementos son menores que cierto elemento especificado. El método `tailSet` obtiene un subconjunto de un objeto `TreeSet` en el que todos sus elementos son mayores o iguales que cierto elemento especificado. Cualquier cambio realizado en el subconjunto se realiza también en el objeto `TreeSet`.
- Los objetos `Map` asocian claves con valores y no pueden contener claves duplicadas. Los objetos `Map` difieren de los objetos `Set` en cuanto a que los primeros contienen claves y valores, mientras que los segundos contienen solamente valo-

res. Los objetos `HashMap` almacenan elementos en una tabla de hash, y los objetos `TreeMap` almacenan elementos en un árbol.

- La interfaz `SortedMap` extiende a `Map` y representa un mapa que mantiene sus claves en forma ordenada. La clase `TreeMap` implementa a `SortedMap`.
- Las colecciones del marco de trabajo de colecciones están desincronizadas. Se utilizan envolturas de sincronización para las colecciones que podrían ser utilizadas por varios subprocesos.
- La API `Collections` proporciona un conjunto de métodos `public static` para convertir las colecciones a versiones no modificables. Las envolturas no modificables lanzan excepciones `UnsupportedOperationException` si se trata de modificar la colección.
- El marco de trabajo de colecciones proporciona varias implementaciones abstractas de interfaces de colecciones, a partir de las cuales el programador puede obtener rápidamente implementaciones personalizadas completas.

TERMINOLOGÍA

`AbstractCollection`, clase
`AbstractList`, clase
`AbstractMap`, clase
`AbstractSequentialList`, clase
`AbstractSet`, clase
`add`, método de `List`
`addFirst`, método de `List`
`addLast`, método de `List`
algoritmos
ArrayList
arreglos
arreglos como colecciones
`asList`, método de `Arrays`
asociación de uno a uno
asociaciones
asociar claves a valores
`binarySearch`, método de `Arrays`
`binarySearch`, método de `Collections`
clase de envoltura
clave
`clear`, método de `List`
cola
colas con dos partes finales (deque)
colección
colección ordenada
colecciones colocadas en arreglos
colecciones modificables
colecciones no modificables
`Collection`, interfaz
`Collections`, clase
`Comparable`, interfaz
comparación lexicográfica
`Comparator`, objeto
`compareTo`, método `Comparable`
`copy`, método de `Collections`
elementos duplicados
eliminar un elemento de la colección
`Enumeration`, interfaz
envolturas de sincronización
`fill`, método de `Arrays`
`fill`, método de `Collections`
`HashMap`, clase
`HashSet`, clase
`hasNext`, método de `Iterator`
insertar un elemento en una colección
`isEmpty`, método de la clase `Map`
iterador
iterador bidireccional
`Iterator`, interfaz
`LinkedList`, clase
`List`, interfaz
`ListIterator`, interfaz
`Map`, interfaz de colección
mapa
marco de trabajo de colecciones
`max`, método de `Collections`
método de comparación natural
métodos de vista de rango
`min`, método de `Collections`
`next`, método de `Iterator`
orden natural
ordenamiento
ordenamiento estable
ordenar un objeto `List`
`reverse`, método de `Collections`
`reverseOrder`, método de `Collections`
secuencia
`Set`, interfaz
`shuffle`, método de `Collections`
`sort`, método de `Arrays`
`sort`, método de `Collections`
`SortedMap`, interfaz de colección
`SortedSet`, interfaz de colección
`TreeMap`, clase
`TreeSet`, clase
ver un arreglo como objeto `List`
vista

EJERCICIOS DE AUTOEVALUACIÓN

22.1 Complete las siguientes oraciones:
a) A los objetos `Object` en una colección se les llama _____.
b) Se puede tener acceso a un elemento en un objeto `List` si se utiliza el _____ de ese elemento.
c) A los objetos `List` algunas veces se les llama _____.
d) Usted puede utilizar una _____ para crear una colección que ofrezca acceso de sólo lectura a otros y le permita a usted el acceso de lectura-escritura.
e) _____ pueden usarse para crear pilas, colas, árboles y colas con dos partes finales (conocidas en inglés como "deque").

22.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Un objeto `Set` puede contener valores duplicados.

b) Un objeto `Map` puede contener claves duplicadas.
c) Un objeto `LinkedList` puede contener valores duplicados.
d) `Collections` es una `interface`.
e) Los objetos `Iterator` pueden eliminar elementos, mientras que los objetos `Enumeration` no.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

22.1 a) Elementos. b) Índice. c) Secuencias. d) Envoltura no modificable. e) Los objetos `LinkedList`.

22.2 a) Falso. Un objeto `Set` no puede contener valores duplicados.
b) Falso. Un objeto `Map` no puede contener claves duplicadas.
c) Verdadero.
d) Falso. `Collections` es una `class`, y `Collection` es una `interface`.
e) Verdadero.

EJERCICIOS

22.3 Defina cada uno de los siguientes términos:
a) `Collection`
b) `Collections`
c) `Comparator`
d) `List`

22.4 Conteste brevemente las siguientes preguntas:
a) ¿Cuál es la principal diferencia entre un objeto `Set` y un objeto `Map`?
b) ¿Puede pasarse un arreglo bidimensional al método `asList` de `Arrays`? De ser así, ¿cómo se accedería a un elemento individual?
c) ¿Qué debe hacer usted antes de agregar un tipo primitivo (por ejemplo, `double`) a una colección?

22.5 Explique brevemente la operación de cada uno de los siguientes métodos relacionados con la clase `Iterator`:
a) `iterator`
b) `hasNext`
c) `next`

22.6 Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Los elementos en un objeto `Collection` deben ordenarse en forma ascendente para que se pueda realizar una búsqueda binaria mediante el método `binarySearch`.
b) El método `first` obtiene el primer elemento en un objeto `TreeSet`.
c) Un objeto `List` creado con el método `asList` de `Arrays` puede cambiar su tamaño.
d) La clase `Arrays` proporciona el método estático `sort` para ordenar los elementos de un arreglo.

22.7 Reescriba el método `imprimirLista` de la figura 22.4 para que utilice un objeto `ListIterator`.

22.8 Reescriba las líneas 14 a 21 en la figura 22.4 para que sea más conciso, utilizando el método `asList` y el constructor `LinkedList` que toma un argumento `Collection`.

22.9 Escriba un programa que lea una serie de nombres de pila y los almacene en un objeto `LinkedList`. No almacene nombres duplicados. Permita al usuario buscar un nombre de pila.

22.10 Modifique el programa de la figura 22.14 para contar el número de ocurrencias de cada letra, en vez de cada palabra. Por ejemplo, la cadena "`HOLA A TODOS`" contienen una `H`, tres `O`s, una `L`, dos `A`s, una `T`, una `D` y una `S`.

22.11 Escriba un programa que determine e imprima el número de palabras duplicadas en una oración. Trate igual a las letras mayúsculas y las minúsculas. Ignore lo signos de puntuación.

22.12 Reescriba su solución al ejercicio 20.8 para utilizar una colección `LinkedList`.

22.13 Reescriba su solución al ejercicio 20.9 para utilizar una colección `LinkedList`.

22.14 Escriba un programa que tome un número entero como entrada por parte de un usuario y determine si este número es primo. Si es primo, agréguelo a un objeto `JTextArea`. Si el número no es primo, muestre los factores primos del número en un objeto `JLabel`. Recuerde que los factores de un número primo son solamente 1 y el número primo en sí. Cada número que no sea primo tiene una factorización única de número primo. Por ejemplo, considere el número 54. Los factores primos de 54 son 2, 3, 3 y 3. Cuando estos valores se multiplican, el resultado es 54. Para el número 54, los factores primos mostrados deben ser 2 y 3. Use objetos `Set` como parte de su solución.

22.15 Reescriba su solución al ejercicio 20.21 para utilizar un objeto `LinkedList`.

22.16 Escriba un programa que utilice un objeto `StringTokenizer` para dividir en tokens una línea de texto introducida por el usuario, y que coloque a cada uno de los tokens en un árbol. Imprima los elementos del árbol ordenado.

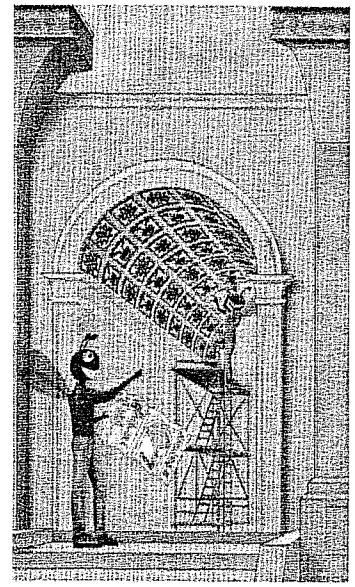
http://librosysolucionarios.net

23

Conectividad de bases de datos en Java con JDBC™

Objetivos

- Comprender las bases de datos relacionales.
- Comprender las consultas básicas de bases de datos mediante el uso de SQL.
- Utilizar las clases e interfaces del paquete `java.sql` para manipular bases de datos.



Es un error capital especular antes de tener datos.
Arthur Conan Doyle

Ahora ven, escríbelo en una tablilla, grábalo en un libro, y que dure hasta el último día, para testimonio.
La Santa Biblia, Isaías 30:8

Primero consiga los hechos, y después podrá distorsionarlos según le parezca.
Mark Twain

Me gustan dos tipos de hombres: domésticos y foráneos.
Mae West

http://librosysolucionarios.net

Plan general

23.1 Introducción
23.2 Bases de datos relacionales
23.3 Generalidades acerca de las bases de datos relacionales: La base de datos `libros`
23.4 SQL
23.4.1 Consulta básica `SELECT`
23.4.2 La cláusula `WHERE`
23.4.3 La cláusula `ORDER BY`
23.4.4 Cómo fusionar datos de varias tablas: `INNER JOIN`
23.4.5 La instrucción `INSERT`
23.4.6 La instrucción `UPDATE`
23.4.7 La instrucción `DELETE`
23.5 Creación de la base de datos `libros` en Cloudscape
23.6 Manipulación de bases de datos con JDBC
23.6.1 Cómo conectarse y realizar consultas en una base de datos
23.6.2 Consultas en la base de datos `libros`
23.7 Procedimientos almacenados
23.8 Recursos en Internet y World Wide Web

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Bibliografía

23.1 Introducción

Una *base de datos* es una colección organizada de datos. Existen diversas estrategias para organizar datos y facilitar el acceso y la manipulación. Un *sistema de administración de bases de datos* (*DBMS*) proporciona los mecanismos para almacenar y organizar datos en una manera consistente con el formato de la base de datos. Los sistemas de administración de bases de datos permiten el acceso y almacenamiento de datos sin necesidad de preocuparse por la su representación interna.

En la actualidad, los sistemas de bases de datos más populares son las *bases de datos relacionales*. Un lenguaje llamado *SQL* es el lenguaje estándar internacional que se utiliza casi universalmente con las bases de datos relacionales para realizar *consultas* (es decir, para solicitar información que satisfaga ciertos criterios) y para manipular datos.

Algunos *sistemas de administración de bases de datos relacionados* (*RDBMS*) populares son Microsoft SQL Server, Oracle, Sybase, DB2, Informix y MySQL. En este capítulo presentaremos ejemplos utilizando *Cloudscape 5.1.2*; un RDBMS de IBM creado completamente en Java. Cloudscape 5.1.2 puede descargarse de `www.ibm.com/software/data/cloudscape`. (*Nota:* Hablaremos sobre las características básicas de Cloudscape que se requieren para ejecutar los ejemplos en este capítulo. Por favor consulte la documentación detallada de Cloudscape para obtener información completa acerca de su uso; también puede consultar en el sitio Web `www.ibacnet.com` para más detalles sobre la instalación .)

Los programas en Java se comunican con las bases de datos y manipulan sus datos utilizando la *API JDBC™*. Un *controlador de JDBC* implementa la interfaz para una base de datos específica. Al separar la API de los controladores específicos, los desarrolladores pueden cambiar la base de datos subyacente sin necesidad de modificar el código de Java para tener acceso a la base de datos. La mayoría de los sistemas de administración de bases de datos populares incluyen ahora controladores de JDBC. También hay muchos controladores de JDBC de terceros disponibles. En este capítulo presentaremos la tecnología JDBC y la emplearemos para manipular una base de datos de Cloudscape. Las técnicas que demostraremos aquí también pueden usarse para manipular otras bases de datos que tengan controladores de JDBC. Consulte la documentación de su sistema para determinar si su DBMS incluye un controlador de JDBC. Incluso si su DBMS no viene con un controlador de JDBC, muchos distribuidores independientes proporcionan estos controladores para una amplia variedad de bases de datos. Para obtener más información acerca de JDBC, visite la página:

```
java.sun.com/products/jdbc/
```

Este sitio contiene información relacionada con JDBC, incluyendo las especificaciones de JDBC, preguntas frecuentes (FAQs) acerca de JDBC, un centro de recursos de aprendizaje, descarga de software y demás información importante. Para obtener una lista de controladores de JDBC disponibles, visite la página:

```
industry.java.sun.com/products/jdbc/drivers/
```

Este sitio proporciona un motor de búsqueda para ayudarlo a localizar los controladores apropiados para su DBMS.

23.2 Bases de datos relacionales

Una *base de datos relacional* es una representación lógica de datos que permite acceder a éstos sin necesidad de considerar la estructura física de estos datos. Una base de datos relacional almacena los datos en *tablas*. En la figura 23.1 se muestra una tabla de ejemplo que podría utilizarse en un sistema de personal. El nombre de la tabla es `Empleado`, y su principal propósito es almacenar los atributos de un empleado. Las tablas están compuestas de *filas*, y las filas, de *columnas* en las que se almacenan los valores. Esta tabla consiste de seis filas. La columna `Numero` de cada fila en esta tabla es la *clave principal* para la tabla. Una clave principal es una columna (o grupo de columnas) en una tabla que tiene un valor único, el cual no puede duplicarse en las demás filas. Esto garantiza que cada fila pueda identificarse por su clave principal. Algunos buenos ejemplos de columnas con clave primaria son un número de Seguro Social, un número de identificación de empleado y un número de pieza en un sistema de inventario, ya que se garantiza que los valores en cada una de esas columnas serán únicos. Las filas de la figura 23.1 se muestran en orden, con base en la clave principal. En este caso, las filas se muestran en orden ascendente; también podríamos utilizar el orden descendente. No se garantiza que las filas en las tablas se almacenen en un orden específico. Como lo demostraremos en un ejemplo más adelante, los programas pueden especificar criterios de ordenamiento al solicitar datos de una base de datos.

Cada columna de la tabla representa un atributo de datos distinto. Las filas generalmente son únicas (por clave primaria) dentro de una tabla, pero los valores de columnas específicas pueden duplicarse entre filas. Por ejemplo, tres filas distintas en la columna `Departamento` de la tabla `Empleado` contienen el número `413`.

A menudo los distintos usuarios de una base de datos se interesan en datos diferentes, y en relaciones distintas entre esos datos. La mayoría de los usuarios requieren solamente de ciertos subconjuntos de las filas y columnas. Para obtener estos subconjuntos, utilizamos instrucciones de SQL para especificar cuáles datos *seleccionar* de una tabla. SQL proporciona un conjunto completo de instrucciones (incluyendo `SELECT`) que permiten a los programadores definir *consultas* complejas para seleccionar datos de una tabla. Por ejemplo, podríamos seleccionar datos de la tabla en la figura 23.1 para crear un resultado que muestre en dónde se ubican

Fila

Numero	Nombre	Departamento	Salario	Ubicacion
23603	Romero	413	1100	Monterrey
24568	Villarreal	413	2000	Monterrey
34589	Gutierrez	642	1800	Distrito Federal
35761	Mendez	611	1400	Guadalajara
47132	Napoles	413	9000	Monterrey
78321	Estebez	611	8500	Guadalajara

Clave principal — Columna

Figura 23.1 Datos de ejemplo para la tabla `Empleado`.

http://librosysolucionarios.net

Departamento	Ubicacion
413	Monterrey
642	Distrito Federal
611	Guadalajara

Figura 23.2 Resultado de seleccionar datos distintos de `Departamento` y `Ubicacion` de la tabla `Empleado`.

los departamentos. Este resultado se muestra en la figura 23.2. Hablaremos sobre las consultas de SQL en la sección 23.4.

23.3 Generalidades acerca de las bases de datos relacionales: La base de datos `libros`

En esta sección veremos las generalidades sobre las bases de datos, y para ello, emplearemos una base de datos llamada `libros`, misma que creamos para este capítulo. Antes de hablar sobre SQL, veremos una descripción general de las tablas de la base de datos `libros`; con el fin de explicar varios conceptos de bases de datos, incluyendo el uso de SQL para obtener información de la base de datos y manipular los datos. En el directorio de ejemplos para este capítulo (en el CD que acompaña al libro) encontrará una secuencia de comandos (script) para crear la base de datos. En la sección 23.5 le explicaremos cómo utilizar esta secuencia de comandos.

La base de datos consiste de cuatro tablas: `autores`, `editoriales`, `isbnAutor` y `titulos`. La tabla `autores` (descrita en la figura 23.3) consta de tres columnas que mantienen el número único de identificación de cada autor, su nombre de pila y apellido. La figura 23.4 contiene datos de ejemplo de la tabla `autores` de la base de datos `libros`.

La tabla `editoriales` (descrita en la figura 23.5) consta de dos columnas que representan el número único de identificación de cada editorial y su nombre. La figura 23.6 contiene los datos de la tabla `editoriales` de la base de datos `libros`.

Columna	Descripción
`idAutor`	El número de identificación (ID) del autor en la base de datos. En la base de datos `libros`, esta columna de enteros se define como *autoincrementada*. Para cada fila insertada en esta tabla, la base de datos incrementa automáticamente el valor de `idAutor` para asegurar que cada fila tenga un `idAutor` único. Esta columna representa la clave principal de la tabla.
`nombrePila`	El nombre de pila del autor (una cadena).
`apellidoPaterno`	El apellido paterno del autor (una cadena).

Figura 23.3 La tabla `autores` de la base de datos `libros`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Figura 23.4 Datos de ejemplo de la tabla `autores`.

http://librosysolucionarios.net

Columna	Descripción
`idEditorial`	El número de identificación (ID) de la editorial en la base de datos. Este entero autoincrementado es la clave principal de la tabla.
`nombreEditorial`	El nombre de la editorial (una cadena).

Figura 23.5 La tabla `editoriales` de la base de datos `libros`.

idEditorial	nombreEditorial
1	Prentice Hall
2	Prentice Hall PTG

Figura 23.6 Datos de la tabla `editoriales`.

Columna	Descripción
`isbn`	El número ISBN del libro (una cadena). La clave principal de la tabla.
`titulo`	Título del libro (una cadena).
`numeroEdicion`	Número de edición del libro (un entero).
`Copyright`	Año de edición (copyright) del libro (una cadena).
`idEditorial`	El número de identificación (ID) de la editorial (un entero). Una clave externa para la tabla `editoriales`.
`archivoImagen`	Nombre del archivo que contiene la imagen de la portada del libro (una cadena).
`precio`	Precio de venta al público sugerido para el libro (un número real). (*Nota:* Los precios que se muestran en este libro son para fines de ejemplo solamente.)

Figura 23.7 La tabla `titulos` de la base de datos `libros`.

La tabla `titulos` (descrita en la figura 23.7) consta de siete columnas que mantienen información general acerca de cada libro en la base de datos, incluyendo el ISBN, título, número de edición, año de edición (copyright), número de identificación (ID) de la editorial, el nombre de un archivo que contiene una imagen de la portada del libro y el precio. La columna `idEditorial` es una *clave externa* (una clave que concuerda con la clave principal en otra tabla; por ejemplo, `idEditorial` en la tabla `editoriales`). Las claves externas se especifican al crear una tabla. La clave externa ayuda a mantener la *Regla de la integridad referencial*. Toda clave externa debe aparecer como la clave principal de otra tabla. Las claves externas permiten que se *unan* filas de varias tablas para fines de analizar los datos. Hay una relación de uno a varios entre una clave principal y su correspondiente clave externa. Esto significa que una clave externa puede aparecer muchas veces en su propia tabla, pero solamente puede aparecer una vez (como la clave principal) en otra tabla. La figura 23.8 contiene datos de ejemplo de la tabla `titulos`.

La tabla `isbnAutor` (descrita en la figura 23.9) consta de dos columnas que representan el número único de identificación de cada editorial y su nombre. Esta tabla asocia a los autores con sus libros. En conjunto, estas claves externas representan la relación entre autores y libros; una fila en la tabla `autores` puede estar asociada con muchas filas en la tabla `titulos` y viceversa. La figura 23.10 contiene los datos de la tabla `isbnAutor` de la base de datos `libros`. ISBN son las siglas de "International Standard Book Number" (Número internacional normalizado para libros); un esquema de numeración utilizado por las editoriales en todo el mundo para dar a cada libro un número de identificación único.

isbn	titulo	numero Edicion	copyright	idEditorial	archivo Imagen	precio
0130895725	C How to Program	3	2001	1	chtp3.jpg	74.95
0130384747	C++ How to Program	4	2002	1	cpphtp4.jpg	74.95
0130461342	Java Web Services for Experienced Programmers	1	2002	1	jwsfep1.jpg	54.95
0131016210	Java How to Program	5	2003	1	jhtp5.jpg	74.95
0130852473	The Complete Java 2 Training Course	5	2002	2	javactc5.jpg	109.95
0130895601	Advanced Java 2 Platform How to Program	1	2002	1	advjhtp1.jpg	74.95

Figura 23.8 Datos de ejemplo para la tabla `titulos` de la base de datos `libros`.

Columna	Descripción
`idAutor`	El número de identificación (ID) del autor, una clave externa para la tabla `autores`.
`isbn`	El ISBN de un libro, una clave externa para la tabla `titulos`.

Figura 23.9 La tabla `isbnAutor` de la base de datos `libros`.

idAutor	isbn	idAutor	isbn
1	0130895725	2	0139163050
2	0130895725	3	0130829293
2	0132261197	3	0130284173
2	0130895717	3	0130284181
2	0135289106	4	0130895601

Figura 23.10 Datos de ejemplo para la tabla `isbnAutor` de la base de datos `libros`.

La figura 23.11 es un *diagrama de relación de entidades (ER)* para la base de datos `libros`. Este diagrama muestra las diversas tablas en la base de datos, así como las relaciones entre ellas. El primer compartimiento en cada cuadro contiene el nombre de la tabla. Los nombres en **negritas** son claves principales. La clave principal de una tabla identifica en forma única a cada fila. Cada fila debe tener un valor en la clave principal, y éste debe ser único en la tabla. A esto se le conoce como "*Regla de integridad de entidades*".



Error común de programación 23.1

Si no se proporciona un valor para cada columna en una clave principal, se quebranta la Regla de integridad de entidades y el DBMS reporta un error.



Error común de programación 23.2

Al proporcionar el mismo valor para la clave principal en varias filas, el DBMS reporta un error.

Las líneas que conectan las tablas en la figura 23.11 representan las *relaciones*. Considere la línea entre las tablas `editoriales` y `titulos`. En el extremo de la línea que va a `editoriales` hay un `1`, y en el extremo que va a `titulos` hay un símbolo de infinito (∞), el cual indica una *relación de uno a varios*; lo que cualquier editorial de la tabla `editoriales` puede tener un número arbitrariamente grande de libros en la ta-

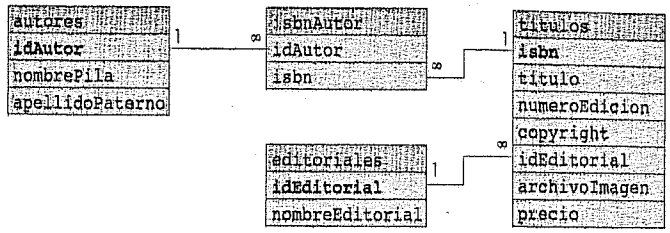


Figura 23.11 Relaciones de las tablas en la base de datos `libros`.

bla `titulos`. Observe que la línea de relación enlaza a la columna `idEditorial` en la tabla `editoriales` con la columna `idEditorial` en la tabla `titulos`. La columna `idEditorial` en la tabla `titulos` es una *clave externa*.



Error común de programación 23.3

Al proporcionar un valor de clave externa que no aparezca como valor de clave principal en otra tabla, se quebranta la Regla de integridad referencial y el DBMS reporta un error.

La línea entre las tablas `isbnAutor` y `autores` indica que, para cada autor en la tabla `autores`, puede haber un número arbitrario de ISBNs para los libros escritos por ese autor en la tabla `isbnAutor`. La columna `idAutor` en la tabla `isbnAutor` es una clave externa que concuerda con la columna `idAutor` (la clave principal) de la tabla `autores`. Observe de nuevo que la línea entre las tablas enlaza a la clave externa en la tabla `isbnAutor` con la correspondiente clave principal en la tabla `autores`. La tabla `isbnAutor` asocia filas en las tablas `titulos` y `autores`.

Por último, la línea entre las tablas `titulos` e `isbnAutor` muestra una relación de uno a varios; un título puede ser escrito por cualquier número de autores. De hecho, el único propósito de la tabla `isbnAutor` es proporcionar una relación de varios a varios entre las tablas `autores` y `titulos`; un autor puede escribir cualquier número de libros y un libro puede tener cualquier número de autores.

23.4 SQL

En esta sección veremos una descripción general acerca de SQL, en el contexto de nuestra base de datos `libros`. En los ejemplos posteriores, usted podrá utilizar las consultas de SQL que describimos aquí.

Las palabras clave de SQL enlistadas en la figura 23.12 se describen en las siguientes subsecciones, dentro del contexto de consultas SQL completas; las demás palabras clave de SQL se encuentran más allá del

Palabra clave de SQL	Descripción
`SELECT`	Recupera datos de una o más tablas.
`FROM`	Las tablas involucradas en la consulta. Se requiere para cada `SELECT`.
`WHERE`	Los criterios de selección que determinan cuáles filas se van a recuperar, eliminar o actualizar.
`GROUP BY`	Criterios para agrupar filas.
`ORDER BY`	Criterios para ordenar filas.
`INNER JOIN`	Fusionar filas de varias tablas.
`INSERT`	Insertar filas en una tabla especificada.
`UPDATE`	Actualizar filas en una tabla especificada.
`DELETE`	Eliminar filas de una tabla especificada.

Figura 23.12 Palabras clave para consultas de SQL.

http://librosysolucionarios.net

alcance de este libro. (*Nota:* Para obtener más información acerca de SQL, consulte los recursos de World Wide Web en la sección 23.8 y la bibliografía al final de este capítulo.)

23.4.1 Consulta básica SELECT

Veamos ahora varias consultas de SQL que extraen información de la base de datos `libros`. Una consulta de SQL "selecciona" filas y columnas de una o más tablas en una base de datos. Dichas selecciones se llevan a cabo mediante *consultas* ***SELECT***. La forma básica de una consulta SELECT es:

```
SELECT * FROM nombreDeTabla
```

En la consulta anterior, el asterisco (`*`) indica que deben recuperarse todas las columnas de la tabla *nombreDeTabla*. Por ejemplo, para recuperar todos los datos en la tabla `autores`, podemos utilizar la siguiente consulta:

```
SELECT * FROM autores
```

Para recuperar sólo ciertas columnas de una tabla, reemplace el asterisco (`*`) con una lista separada por comas de los nombres de las columnas. Por ejemplo, para recuperar solamente las columnas `idAutor` y `apellidoPaterno` para todas las filas en la tabla `autores`, utilice la siguiente consulta:

```
SELECT idAutor, apellidoPaterno FROM autores
```

Esta consulta devuelve los datos que se muestran en la figura 23.13.

Observación de ingeniería de software 23.1

Para la mayoría de las instrucciones de SQL, no debe utilizarse el asterisco (`*`) para especificar nombres de columnas. En general, los programadores procesan los resultados sabiendo de antemano el orden de las columnas en el resultado; por ejemplo, al seleccionar `idAutor` y `apellidoPaterno` de la tabla `autores` nos aseguramos que las columnas aparezcan en el resultado con `idAutor` como la primera columna y `apellidoPaterno` como la segunda. Generalmente los programas procesan las columnas de resultado especificando el número de columna en el resultado (empezando con el número 1 para la primera columna).

Observación de ingeniería de software 23.2

Al especificar los nombres de columnas que se van a seleccionar, se garantiza que las columnas se devuelvan siempre en el orden especificado y también se evita que se devuelvan columnas innecesarias, incluso si cambia el orden real.

Error común de programación 23.4

Si un programador supone que al utilizar el asterisco (`*`) las columnas, en un resultado, se devolverán siempre en el mismo orden, el programa podría procesar el resultado incorrectamente. Si cambia el orden de las columnas en la(s) tabla(s), el orden de las columnas en el resultado cambia de manera acorde.

Tip de rendimiento 23.1

Si se desconoce el orden de las columnas en un resultado, un programa debe procesarlas por su nombre. Al especificar los nombres de las columnas que se van a seleccionar, se permite a la aplicación que recibe el resultado conocer de antemano el orden de las columnas. En este caso, el programa puede procesar los datos con más eficiencia, ya que se puede acceder a las columnas mediante el número de columna.

idAutor	apellidoPaterno
1	Deitel
2	Deitel
3	Nieto
4	Santry

Figura 23.13 Datos de ejemplo para `idAutor` y `apellidoPaterno` de la tabla `autores`.

23.4.2 La cláusula WHERE

En la mayoría de los casos es necesario localizar, en una base de datos, filas que cumplan con ciertos *criterios de selección* (formalmente llamados *predicados*). SQL utiliza la *cláusula* ***WHERE*** en una consulta `SELECT` para especificar los criterios de selección para la consulta. La forma básica de una consulta `SELECT` con criterios de selección es:

```
SELECT nombreDeColumna1, nombreDeColumna2, ... FROM nombreDeTabla WHERE criterios
```

Por ejemplo, para seleccionar las columnas `titulo`, `numeroEdicion` y `copyright` de la tabla `titulos` para las cuales la fecha de `copyright` sea mayor que `2000`, utilice la siguiente consulta:

```
SELECT titulo, numeroEdicion, copyright
   FROM titulos
   WHERE copyright > 2000
```

En la figura 23.14 se muestra el resultado de la consulta anterior.

Los criterios de la cláusula `WHERE` pueden contener los operadores `<`, `>`, `<=`, `>=`, `=`, `<>` y `LIKE` El operador `LIKE` se utiliza para *relacionar patrones* con los caracteres comodines *porcentaje* (`%`) y *guión bajo* (`_`). El relacionar patrones permite a SQL buscar cadenas que concuerden con un patrón dado.

Un patrón que contenga un carácter de porcentaje (`%`) busca cadenas que tengan cero o más caracteres en la posición del carácter de porcentaje en el patrón. Por ejemplo, la siguiente consulta localiza las filas de todos los autores cuyo apellido paterno empiece con la letra `D`:

```
SELECT idAutor, nombrePila, apellidoPaterno
   FROM autores
   WHERE apellidoPaterno LIKE 'D%'
```

La consulta anterior selecciona las dos filas que se muestran en la figura 23.15, ya que dos de los cuatro autores en nuestra base de datos tienen un apellido paterno que empieza con la letra `D` (seguida de cero o más caracteres). El signo de `%` y el operador `LIKE` de la cláusula `WHERE` indican que puede aparecer cualquier número de caracteres después de la letra `D` en la columna `apellidoPaterno`. Observe que la cadena del operador está encerrada entre caracteres de comilla sencilla.

titulo	numeroEdicion	copyright
C How to Program	3	2001
C++ How to Program	4	2002
The Complete C++ Training Course	4	2002
Internet and World Wide Web How to Program	2	2002
Java How to Program	5	2003
XML How to Program	1	2001
Perl How to Program	1	2001
Advanced Java 2 Platform How to Program	1	2002

Figura 23.14 Ejemplos de títulos, de la tabla `titulos`, con fechas de copyright posteriores a 2000.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel

Figura 23.15 Autores, de la tabla `autores`, cuyo apellido paterno empieza con `D`.

http://librosysolucionarios.net



Tip de portabilidad 23.1

Consulte la documentación de su sistema de bases de datos para determinar si SQL es susceptible al uso de mayúsculas en su sistema, y para determinar la sintaxis de las palabras clave de SQL (por ejemplo, ¿deben estar completamente en mayúsculas, completamente en minúsculas o puede ser una combinación de ambas?).



Tip de portabilidad 23.2

No todos los sistemas de bases de datos soportan el operador `LIKE`*, por lo que debe asegurarse de leer cuidadosamente la documentación de su sistema de bases de datos.*



Tip de portabilidad 23.3

Algunas bases de datos utilizan el carácter `*` *en lugar del carácter* `%` *en un patrón.*

Un guión bajo (_) en la cadena del patrón indica un carácter comodín individual en esa posición. Por ejemplo, la siguiente consulta localiza las filas de todos los autores cuyo apellido paterno empiece con cualquier carácter (lo que se especifica con _), seguido por la letra `i`, seguida por cualquier número de caracteres adicionales (lo que se especifica con `%`):

```
SELECT idAutor, nombrePila, apellidoPaterno
   FROM autores
   WHERE apellidoPaterno LIKE '_i%'
```

La consulta anterior produce la fila que se muestra en la figura 23.16, ya que sólo un autor en nuestra base de datos tiene un apellido paterno que contiene la letra `i` como su segunda letra.



Tip de portabilidad 23.4

Algunas bases de datos utilizan el carácter `?` *en lugar del carácter* `_` *en un patrón.*

23.4.3 La cláusula `ORDER BY`

El resultado de una consulta puede ordenarse en forma ascendente o descendente, mediante el uso de la *cláusula* `ORDER BY` opcional. La forma básica de una instrucción `SELECT` con una cláusula `ORDER BY` es:

```
SELECT nombreDeColumna1, nombreDeColumna2 , ... FROM nombreDeTabla ORDER BY columna ASC
SELECT nombreDeColumna1, nombreDeColumna2 , ... FROM nombreDeTabla ORDER BY columna DESC
```

en donde `ASC` especifica el orden ascendente (de menor a mayor), `DESC` especifica el orden descendente (de mayor a menor) y *columna* especifica la columna en la cual se basa el ordenamiento. Por ejemplo, para obtener la lista de autores en orden ascendente por apellido paterno (figura 23.17), utilice la siguiente consulta:

```
SELECT idAutor, nombrePila, apellidoPaterno
   FROM autores
   ORDER BY apellidoPaterno ASC
```

Observe que el orden predeterminado es ascendente, por lo que `ASC` es opcional. Para obtener la misma lista de autores en orden descendente por apellido paterno (figura 23.18), utilice la siguiente consulta:

```
SELECT idAutor, nombrePila, apellidoPaterno
   FROM autores
   ORDER BY apellidoPaterno DESC
```

idAutor	nombrePila	apellidoPaterno
3	Tem	Nieto

Figura 23.16 El único autor de la tabla `autores` cuyo apellido paterno contiene a `i` como segunda letra.

idAutor	nombrePila	apellidoPaterno
2	Paul	Deitel
1	Harvey	Deitel
3	Tem	Nieto
4	Sean	Santry

Figura 23.17 Datos de ejemplo de la tabla `autores` ordenados en forma ascendente por la columna `apellidoPaterno`.

idAutor	nombrePila	apellidoPaterno
4	Sean	Santry
3	Tem	Nieto
2	Paul	Deitel
1	Harvey	Deitel

Figura 23.18 Datos de ejemplo de la tabla `autores` ordenados en forma descendente por la columna `apellidoPaterno`.

Pueden usarse varias columnas para ordenar mediante una cláusula `ORDER BY`, de la siguiente forma:

```
ORDER BY columna1 tipoDeOrden, columna2 tipoDeOrden, ...
```

en donde *tipoDeOrden* puede ser `ASC` o `DESC`. Observe que el *tipoDeOrden* no tiene que ser idéntico para cada columna. La consulta:

```
SELECT idAutor, nombrePila, apellidoPaterno
   FROM autores
   ORDER BY apellidoPaterno, nombrePila
```

ordena en forma ascendente todas las filas por apellido paterno, y después por nombre de pila. Si varias filas tienen el mismo valor de apellido paterno, se devuelven ordenadas por nombre de pila (figura 23.19).

Las cláusulas `WHERE` y `ORDER BY` pueden combinarse en una consulta. Por ejemplo, la consulta:

```
SELECT isbn, titulo, numeroEdicion, copyright, precio
   FROM titulos
   WHERE titulo
   LIKE '%How to Program'
   ORDER BY titulo ASC
```

devuelve el `isbn`, `titulo`, `numeroEdicion`, `copyright` y `precio` de cada libro en la tabla `titulos` que tenga un `titulo` que termine con `"How to Program"` y los ordena en forma ascendente, por `titulo`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Figura 23.19 Datos de ejemplo de autores de la tabla `autores` ordenados de manera ascendente, por las columnas `apellidoPaterno` y `nombrePila`.

http://librosysolucionarios.net

isbn	titulo	numeroEdicion	copyright	precio
0130895601	Advanced Java 2 Platform How to Program	1	2002	74.95
0130895725	C How to Program	3	2001	74.95
0130384747	C++ How to Program	4	2002	74.95
0130308978	Internet and World Wide Web How to Program	2	2002	74.95
0130284181	Perl How to Program	1	2001	74.95
0134569555	Visual Basic 6 How to Program	1	1999	74.95
0130284173	XML How to Program	1	2001	74.95
013028419x	e-Business and e-Commerce How to Program	1	2001	74.95

Figura 23.20 Ejemplos de libros de la tabla `titulos` cuyos títulos terminan con `How to Program`, y están ordenados en forma ascendente por medio de la columna `titulo`.

El resultado de la consulta se muestra en la figura 23.20. Observe que el título "e-Business and e-Commerce How to Program" aparece al final de la lista, ya que Cloudscape utiliza los valores numéricos Unicode de los caracteres para fines de comparación. Recuerde que las letras minúsculas tienen valores numéricos mayores que las letras mayúsculas.

23.4.4 Cómo fusionar datos de varias tablas: `INNER JOIN`

Los diseñadores de bases de datos a menudo dividen los datos en tablas separadas para asegurarse de no guardar información redundante. Por ejemplo, la base de datos `libros` tiene las tablas `autores` y `titulos`; y utilizamos una tabla `isbnAutor` para almacenar los datos de la relación entre los autores y sus títulos. Si no separáramos esta información en tablas individuales, tendríamos que incluir la información del autor con cada entrada en la tabla `titulos`. Esto implicaría almacenar información duplicada sobre los autores en la base de datos, para quienes hayan escrito varios libros. A menudo es necesario fusionar datos de varias tablas en un solo resultado. Este proceso, que se le conoce como *unir* las tablas, se especifica mediante un operador `INNER JOIN` en la consulta `SELECT`. Un operador `INNER JOIN` fusiona las filas de dos tablas al relacionar los valores en columnas que sean comunes para las dos tablas. La forma básica de un `INNER JOIN` es:

```
SELECT nombreDeColumna1, nombreDeColumna2, ...
FROM tabla1
INNER JOIN tabla2
   ON tabla1.nombreDeColumna = tabla2.nombreDeColumna
```

La cláusula `ON` de `INNER JOIN` especifica las columnas de cada tabla que se comparan para determinar cuáles filas se fusionan. Por ejemplo, la siguiente consulta produce una lista de autores acompañada por los ISBNs para los libros escritos por cada autor.

```
SELECT nombrePila, apellidoPaterno, isbn
FROM autores
INNER JOIN isbnAutor
   ON autores.idAutor = isbnAutor.idAutor
ORDER BY apellidoPaterno, nombrePila
```

La consulta fusiona los datos de las columnas `nombrePila` y `apellidoPaterno` de la tabla `autores` con la columna `isbn` de la tabla `isbnAutor`, ordenando el resultado en forma ascendente por `apellidoPaterno` y `nombrePila`. Observe el uso de la sintaxis *nombreDeTabla.nombreDeColumna* en la cláusula `ON`. Esta sintaxis (a la que se le conoce como *nombre calificado*) especifica las columnas de cada tabla que deben compararse para unir las tablas. La sintaxis "*nombreDeTabla.*" es requerida si las columnas tienen el mismo nombre en ambas tablas. La misma sintaxis puede utilizarse en cualquier consulta para diferenciar entre columnas de tablas distintas que tengan el mismo nombre. En algunos sistemas pueden utilizarse nombres de tablas calificados con el nombre de la base de datos para realizar consultas entre varias bases de datos.

nombrePila	apellidoPaterno	isbn	nombrePila	apellidoPaterno	isbn
Harvey	Deitel	0130895601	Paul	Deitel	0130895717
Harvey	Deitel	0130284181	Paul	Deitel	0132261197
Harvey	Deitel	0134569555	Paul	Deitel	0130895725
Harvey	Deitel	0139163050	Paul	Deitel	0130829293
Harvey	Deitel	0135289106	Paul	Deitel	0134569555
Harvey	Deitel	0130895717	Paul	Deitel	0130829277
Harvey	Deitel	0130284173	Tem	Nieto	0130161438
Harvey	Deitel	0130829293	Tem	Nieto	013028419x
Paul	Deitel	0130852473	Sean	Santry	0130895601

Figura 23.21 Ejemplo de datos de autores e ISBNs para los libros que han escrito, en orden ascendente por `apellidoPaterno` y `nombrePila`.



Observación de ingeniería de software 23.3

Si una instrucción de SQL incluye columnas de varias tablas que tengan el mismo nombre, en la instrucción se debe anteponer a los nombres de esas columnas los nombres de sus tablas y el operador punto (por ejemplo, `autores.idAutor`).



Error común de programación 23.5

En una consulta, si no se califican los nombres para las columnas que tengan el mismo nombre en dos o más tablas se produce un error.

Como siempre, la consulta puede contener una cláusula `ORDER BY`. En la figura 23.21 se muestran los datos del resultado de la consulta anterior, ordenados por `apellidoPaterno` y `nombrePila`. (*Nota:* Para ahorrar espacio dividimos el resultado de la consulta en dos columnas, cada una de las cuales contiene a las columnas `nombrePila`, `apellidoPaterno` e `isbn`.)

23.4.5 La instrucción `INSERT`

La instrucción `INSERT` inserta una fila en una tabla. La forma básica de esta instrucción es:

```
INSERT INTO nombreDeTabla ( nombreDeColumna1, nombreDeColumna2, ..., nombreDeColumnaN )
   VALUES ( valor1, valor2, ..., valorN )
```

en donde *nombreDeTabla* es la tabla en la que se va a insertar la fila. El *nombreDeTabla* va seguido de una lista separada por comas de nombres de columnas entre paréntesis (esta lista no es requerida si la operación `INSERT` especifica un valor para cada columna de la tabla en el orden correcto). La lista de nombres de columnas va seguida por la palabra clave `VALUES` de SQL, y una lista separada por comas de valores entre paréntesis. Los valores especificados aquí deben concordar con las columnas especificadas después del nombre de la tabla, tanto en orden como en tipo (por ejemplo, si *nombreDeColumna1* se supone que debe ser la columna `nombrePila`, entonces *valor1* debe ser una cadena entre comillas sencillas que represente el nombre de pila). Siempre debe enlistar explícitamente las columnas al insertar filas. Si el orden de las columnas cambia en la tabla, al utilizar solamente `VALUES` se puede provocar un error. La instrucción `INSERT`:

```
INSERT INTO autores ( nombrePila, apellidoPaterno )
   VALUES ( 'Alejandra', 'Villarreal' )
```

inserta una fila en la tabla `autores`. La instrucción indica que se proporcionan valores para las columnas `nombrePila` y `apellidoPaterno`. Los valores correspondientes son `'Alejandra'` y `'Villarreal'`. No especificamos un `idAutor` en este ejemplo, ya que `idAutor` es una *columna autoincrementada* en la tabla `autores`. Para cada fila que se agrega a esta tabla, Cloudscape asigna un valor de `idAutor` único que es el siguiente valor en la secuencia autoincrementada (por ejemplo: 1, 2, 3 y así sucesivamente). En este caso,

http://librosysolucionarios.net

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Alejandra	Villarreal

Figura 23.22 Datos de ejemplo de la tabla `autores` después de una operación `INSERT`.

Alejandra Villarreal recibiría el número 5 para `idAutor`. En la figura 23.22 se muestra la tabla `autores` después de la operación `INSERT`.



Error común de programación 23.6

Es un error especificar un valor para una columna que se autoincrementa.



Error común de programación 23.7

Las instrucciones de SQL utilizan el carácter de comilla sencilla (') como delimitador para las cadenas. Para especificar una cadena que contenga una comilla sencilla (como O'Malley) en una instrucción de SQL, la cadena debe tener dos comillas sencillas en la posición en la que aparezca el carácter de comilla sencilla en la cadena (por ejemplo, `'O''Malley'`). El primero de los dos caracteres de comilla sencilla actúa como carácter de escape para el segundo. Si no se utiliza el carácter de escape en una cadena que sea parte de una instrucción de SQL, se produce un error de sintaxis de SQL.

23.4.6 La instrucción `UPDATE`

Una instrucción ***`UPDATE`*** modifica los datos en una tabla. La forma básica de la instrucción `UPDATE` es:

```
UPDATE nombreDeTabla
   SET nombreDeColumna1 = valor1, nombreDeColumna2 = valor2, ..., nombreDeColumnaN = valorN
   WHERE criterios
```

en donde *nombreDeTabla* es la tabla que se va a actualizar. El *nombreDeTabla* va seguido por la palabra clave ***`SET`*** y una lista separada por comas de los pares nombre/valor de las columnas, en el formato *nombreDeColumna=valor*. La cláusula `WHERE` proporciona los criterios que determinan cuáles filas se van a actualizar. La instrucción `UPDATE`:

```
UPDATE autores
   SET apellidoPaterno = 'Garcia'
   WHERE apellidoPaterno = 'Villarreal' AND nombrePila = 'Alejandra'
```

actualiza una fila en la tabla `autores`. La instrucción indica que `apellidoPaterno` recibirá el valor `Garcia` para la fila en la que `apellidoPaterno` sea igual a `Villarreal` y `nombrePila` sea igual a `Alejandra`. (*Nota:* Si hay varias filas con el mismo nombre de pila "Alejandra" y el apellido paterno "Villarreal", esta instrucción modificará a todas esas filas para que tengan el apellido paterno "Garcia"). Si conocemos el `idAutor` desde antes de realizar la operación `UPDATE` (tal vez porque lo hayamos buscado con anterioridad), la cláusula `WHERE` podría quedar de la siguiente manera:

```
WHERE idAutor = 5
```

En la figura 23.23 se muestra la tabla `autores` después de realizar la operación `UPDATE`.

23.4.7 La instrucción `DELETE`

Una instrucción ***`DELETE`*** de SQL elimina filas de una tabla. La forma básica de una instrucción `DELETE` es:

```
DELETE FROM nombreDeTabla WHERE criterios
```

http://librosysolucionarios.net

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Alejandra	Garcia

Figura 23.23 Datos de ejemplo de la tabla `autores` después de una operación `UPDATE`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Figura 23.24 Datos de ejemplo de la tabla `autores` después de una operación `DELETE`.

en donde *nombreDeTabla* es la tabla de la que se van a eliminar filas. La cláusula `WHERE` especifica los criterios utilizados para determinar cuáles filas eliminar. La instrucción `DELETE`:

```
DELETE FROM autores
   WHERE apellidoPaterno = 'Garcia' AND nombrePila = 'Alejandra'
```

elimina la fila de Alejandra Garcia en la tabla `autores`. Si conocemos el `idAutor` desde antes de realizar la operación `DELETE`, la cláusula `WHERE` puede simplificarse de la siguiente manera:

```
WHERE idAutor = 5
```

En la figura 23.24 se muestra la tabla `autores` después de realizar la operación `DELETE`.

23.5 Creación de la base de datos `libros` en Cloudscape

El CD que acompaña a este libro incluye Cloudscape 5.1.2; un sistema de administración de bases de datos incrustado de IBM, creado completamente en Java. Puede obtener información completa acerca de Cloudscape en la siguiente página Web:

```
www.ibm.com/software/data/cloudscape
```

Siga las instrucciones proporcionadas para instalar Cloudscape; éste se ejecuta en muchas plataformas, incluyendo Windows, Solaris, Linux, Macintosh y otros. Para obtener una lista completa de las plataformas en las que se ha probado Cloudscape, visite la página Web:

```
www.ibm.com/software/data/cloudscape/requirements.html
```

Debe haber varios archivos JAR de Cloudscape en la ruta de clases de su programa para que pueda crear y manipular bases de datos en Cloudscape. Para incluir los archivos JAR apropiados en la ruta de clases, ejecute el archivo de procesamiento por lotes `setCP.bat` (o la secuencia de comandos de shell `setCP.ksh` para UNIX) del directorio `Cloudscape_5.1\frameworks\embedded\bin`.

Para cada base de datos de Cloudscape que veremos en este libro, proporcionaremos una secuencia de comandos de SQL para configurar tanto la base de datos como sus tablas. Estas secuencias de comandos pueden ejecutarse mediante una herramienta interactiva de línea de comandos llamada `ij`, la cual forma parte de

Cloudscape. Proporcionamos también un archivo de procesamiento por lotes (`crearBaseDeDatos.bat`) que puede utilizar para iniciar `ij` y ejecutar las secuencias de comandos de SQL. En el directorio de ejemplos para este capítulo, que viene en el CD que acompaña a este libro, encontrará las secuencias de comandos `crearBaseDeDatos` y la secuencia de comandos SQL `libros.sql`. Para crear la base de datos `libros`, primero ejecute el archivo de procesamiento por lotes `setCP.bat` (o la secuencia de comandos de shell `setCP.ksh` para UNIX) del directorio `Cloudscape_5.\bin`. Con esto se establecen las variables de entorno requeridas por nuestra secuencia de comandos `crearBaseDeDatos`. A continuación, vaya al directorio en donde instaló Cloudscape en su equipo (por ejemplo, `C:\Cloudscape_5.1`) y escriba:

```
crearBaseDeDatos libros.sql
```

para ejecutar la secuencia de comandos de SQL. Después de completar esta tarea, se creará en el directorio actual un nuevo directorio con la información sobre la base de datos. Ahora está usted listo para continuar con el primer ejemplo de JDBC. (*Nota:* Escribimos esta secuencia de comandos de manera que pueda ejecutarla de nuevo en cualquier momento, para restaurar el contenido original de la base de datos. La primera vez que ejecute esta secuencia de comandos, generará cuatro mensajes de error al tratar de eliminar las cuatro tablas de la base de datos `libros`. Esto ocurre debido a que la base de datos no existe, por lo que no hay tablas que puedan eliminarse. Simplemente ignore estos mensajes.)

23.6 Manipulación de bases de datos con JDBC

En esta sección presentaremos dos ejemplos que le enseñarán cómo conectarse a una base de datos, hacer consultas en ella y mostrar el resultado de las consultas.

23.6.1 Cómo conectarse y realizar consultas en una base de datos

El ejemplo de la figura 23.25 realiza una consulta simple en la base de datos `libros` para recuperar toda la tabla `autores` y mostrar los datos en un objeto `JTextArea`. El programa muestra cómo conectarse a la base de datos, hacer una consulta en la misma y procesar el resultado. En la siguiente discusión presentaremos los aspectos clave del programa relacionados con JDBC. (*Nota:* En la sección 23.5 se muestra cómo preparar la base de datos de Cloudscape y cómo crear la base de datos `libros`. Los pasos en la sección 23.5 deben realizarse antes de ejecutar el programa de la figura 23.25.)

En la línea 5 se importa el paquete `java.sql`, el cual contiene las clases e interfaces para la API JDBC. En la línea 12 se declara una constante `String` que contiene el nombre de la clase del controlador JDBC para la base de datos de Cloudscape. El programa utilizará este valor para cargar el controlador apropiado en memoria. En la línea 13 se declara una constante de cadena para el URL de la base de datos. Esta constante identifica el nombre de la base de datos a la que nos conectaremos, así como información acerca del protocolo utilizado por el controlador JDBC. El constructor `MostrarAutores` (líneas 22 a 108) se conecta a la base de datos `libros`, realiza una consulta en la base de datos, muestra el resultado de esa consulta y cierra la conexión a la base de datos.

```
1  // Fig. 23.26: MostrarAutores.java
2  // Mostrar el contenido de la tabla autores.
3
4  import java.awt.*;
5  import java.sql.*;
6  import java.util.*;
7  import javax.swing.*;
8
9  public class MostrarAutores extends JFrame {
10
11    // nombre del controlador JDBC y URL de la base de datos
```

Figura 23.25 Cómo mostrar la tabla `autores` de la base de datos `libros`. (Parte 1 de 4.)


```
12    static final String CONTROLADOR_JDBC = "com.ibm.db2.jdbc.DB2Driver";
13    static final String URL_BASEDEDATOS = "jdbc:db2:libros";
14
15    // declarar objetos Connection y Statement para acceder a
16    // la base de datos y realizar consultas
17    private Connection conexion;
18    private Statement instruccion;
19
20    // el constructor se conecta a la base de datos, realiza la consulta, procesa
21    // los resultados y los muestra en la ventana
22    public MostrarAutores()
23    {
24       super( "Tabla autores de la base de datos libros" );
25
26       // conectarse a la base de datos libros y consultar la base de datos
27       try {
28
29          // especificar la ubicación de la base de datos en el sistema de archivos
30          System.setProperty( "db2j.system.home", "C:/Cloudscape_5.1" );
31
32          // cargar clase de controlador de base de datos
33          Class.forName( CONTROLADOR_JDBC );
34
35          // establecer conexión a la base de datos
36          conexion = DriverManager.getConnection( URL_BASEDEDATOS );
37
38          // crear objeto Statement para consultar la base de datos
39          instruccion = conexion.createStatement();
40
41          // consultar la base de datos
42          ResultSet conjuntoResultados =
43             instruccion.executeQuery( "SELECT * FROM autores" );
44
45          // procesar los resultados de la consulta
46          StringBuffer resultados = new StringBuffer();
47          ResultSetMetaData metaDatos = conjuntoResultados.getMetaData();
48          int numeroDeColumnas = metaDatos.getColumnCount();
49
50          for ( int i = 1; i <= numeroDeColumnas; i++ )
51             resultados.append( metaDatos.getColumnName( i ) + "\t" );
52
53          resultados.append( "\n" );
54
55          while ( conjuntoResultados.next() ) {
56
57             for ( int i = 1; i <= numeroDeColumnas; i++ )
58                resultados.append( conjuntoResultados.getObject( i ) + "\t" );
59
60             resultados.append( "\n" );
61          }
62
63          // configurar la GUI y ventana para mostrar el resultado
64          JTextArea areaTexto = new JTextArea( resultados.toString() );
65          Container contenedor = getContentPane();
66
```

http://librosysolucionarios.net

Figura 23.25 Cómo mostrar la tabla `autores` de la base de datos `libros`. (Parte 2 de 4.)

```
67          contenedor.add( new JScrollPane( areaTexto ) );
68
69          setSize( 320, 130 );  // establecer el tamaño de la ventana
70          setVisible( true );   // mostrar la ventana
71
72       }  // fin de bloque try
73
74       // detectar posibles problemas al interactuar con la base de datos
75       catch ( SQLException excepcionSql ) {
76          JOptionPane.showMessageDialog( null, excepcionSql.getMessage(),
77             "Error en base de datos", JOptionPane.ERROR_MESSAGE );
78
79          System.exit( 1 );
80       }
81
82       // detectar posibles problemas al cargar el controlador de la base de datos
83       catch ( ClassNotFoundException claseNoEncontrada ) {
84          JOptionPane.showMessageDialog( null, claseNoEncontrada.getMessage(),
85             "No se encontró el controlador", JOptionPane.ERROR_MESSAGE );
86
87          System.exit( 1 );
88       }
89
90       // asegurar que instrucción y conexión se cierren correctamente
91       finally {
92
93          try {
94             instruccion.close();
95             conexion.close();
96          }
97
98          // manejar posibles excepciones al cerrar instruccion y conexion
99          catch ( SQLException excepcionSql ) {
100            JOptionPane.showMessageDialog( null,
101               excepcionSql.getMessage(), "Error en base de datos",
102               JOptionPane.ERROR_MESSAGE );
103
104            System.exit( 1 );
105         }
106      }
107
108   }  // fin del constructor de MostrarAutores
109
110   // iniciar la aplicación
111   public static void main( String args[] )
112   {
113      MostrarAutores ventana = new MostrarAutores();
114      ventana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
115   }
116
117 }  // fin de la clase MostrarAutores
```

Figura 23.25 Cómo mostrar la tabla **autores** de la base de datos **libros**. (Parte 3 de 4.)

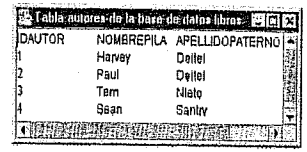


Figura 23.25 Cómo mostrar la tabla **autores** de la base de datos **libros**. (Parte 4 de 4.)

El programa debe cargar el controlador de bases de datos para poder conectarse a la base de datos. En la línea 33 se carga la clase para el controlador de bases de datos. Esta línea lanza una excepción verificada del tipo `java.lang.ClassNotFoundException` si el cargador de clases no puede localizar la clase del controlador. El controlador JDBC de Cloudscape determina las ubicaciones de los archivos de bases de datos, utilizando la propiedad del sistema `db2j.system.home`. En la línea 30 se establece esta propiedad en la ubicación en la que creamos la base de datos `libros`.

JDBC soporta cuatro categorías de controladores: *controlador puente JDBC a ODBC* (tipo 1), *controlador nativo de API desarrollado parcialmente en Java* (tipo 2); *controlador JDBC-Net desarrollado puramente en Java* (tipo 3) y el *controlador de protocolo nativo desarrollado puramente en Java* (tipo 4). En la figura 23.26 se muestra una descripción de cada tipo de controlador. El controlador de Cloudscape `com.ibm.db2j.jdbc.DB2j-Driver` es un controlador de tipo 4.



Observación de ingeniería de software 23.4

La mayoría de los distribuidores de bases de datos proporcionan sus propios controladores de bases de datos JDBC, y muchos distribuidores independientes proporcionan también controladores JDBC. Para obtener más información acerca de los controladores de JDBC, visite el sitio Web sobre JDBC de Sun Microsystems en `java.sun.com/products/jdbc`.

Tipo	Descripción
1	El *controlador puente JDBC a ODBC* conecta los programas de Java con orígenes de datos de Microsoft ODBC (Conectividad abierta de bases de datos). El Kit de desarrollo de software para Java 2 de Sun Microsystems, Inc. Incluye el controlador puente JDBC a ODBC (`sun.jdbc.odbc.JdbcOdbcDriver`). Este controlador por lo general requiere que el controlador ODBC esté instalado en el equipo cliente, y normalmente es necesario que se configure el origen de datos ODBC. El controlador puente se introdujo principalmente para fines de desarrollo, por lo cual no debe utilizarse en aplicaciones de producción.
2	Los *controladores nativos de API desarrollados parcialmente en Java* permiten a los programas de JDBC utilizar APIs para bases de datos específicas (generalmente escritas en C o C++), las cuales permiten a los programas clientes utilizar bases de datos mediante la Interfaz nativa de Java (Java Native Interface). Este tipo de controlador traduce el código JDBC en código de una base de datos específica. Los controladores de tipo 2 se introdujeron por razones similares al controlador puente ODBC tipo 1.
3	Los *controladores JDBC-Net desarrollados puramente en Java* toman las peticiones de JDBC y las traducen en un protocolo de red que no es para una base de datos específica. Estas peticiones se envían a un servidor, el cual traduce las peticiones de la base de datos en un protocolo específico para esa base de datos.
4	Los *controladores de protocolo nativo desarrollados puramente en Java* convierten las peticiones de JDBC en protocolos de red para bases de datos específicas, de manera que los programas de Java puedan conectarse directamente con una base de datos.

Figura 23.26 Tipos de controladores JDBC.

http://librosysolucionarios.net

Observación de ingeniería de software 23.5

En la plataforma Microsoft Windows, la mayoría de las bases de datos soportan el acceso mediante la Conectividad abierta de bases de datos (ODBC). Ésta es una tecnología desarrollada por Microsoft para permitir el acceso genérico a diversos sistemas de bases de datos en la plataforma Windows (y en algunas plataformas UNIX). El Kit de desarrollo de software para Java 2 (J2SDK) incluye el controlador de bases de datos puente JDBC a ODBC, para permitir que cualquier programa de Java pueda acceder a cualquier origen de datos ODBC. El controlador está compuesto por la clase `JdbcOdbcDriver` *en el paquete* `sun.jdbc.odbc`.

En la línea 36 de la figura 23.25 se crea un objeto `Connection` (paquete `java.sql`), el cual es referenciado mediante `conexion`. Un objeto que implementa a la interfaz `Connection` administra la conexión entre el programa de Java y la base de datos. Los objetos `Connection` permiten a los programas crear instrucciones de SQL para manipular bases de datos. El programa inicializa a `conexion` con el resultado de una llamada al método `static` `getConnection` de la clase `DriverManager` (paquete `java.sql`), el cual trata de conectarse a la base de datos especificada en su argumento URL. El URL localiza la base de datos (posiblemente en una red o en el sistema local de archivos de la computadora). El URL `jdbc:db2j:libros` especifica el protocolo de comunicación (`jdbc`), el *subprotocolo* de comunicación (`db2j`) y el nombre de la base de datos (`libros`). El subprotocolo `db2j` indica que el programa utiliza un subprotocolo específico de IBM para conectarse a la base de datos de Cloudscape. Si el objeto `DriverManager` no puede conectarse a la base de datos, el método `getConnection` lanza una excepción `SQLException` (paquete `java.sql`).

Observación de ingeniería de software 23.6

La mayoría de los sistemas de administración de bases de datos requieren que el usuario inicie la sesión para poder administrar el contenido de la base de datos. El método `getConnection` *de* `DriverManager` *está sobrecargado con versiones que permiten al programa proporcionar el nombre de usuario y la contraseña para obtener acceso.*

En la línea 39 se invoca el método `createStatement` de `Connection` para obtener un objeto que implemente a la interfaz `Statement` (paquete `java.sql`). El programa utiliza al objeto `Statement` para enviar instrucciones de SQL a la base de datos.

En las líneas 42 y 43 se utiliza el método `executeQuery` de `Statement` para enviar una consulta que seleccione toda la información sobre los autores en la tabla `autores`. Este método devuelve un objeto que implementa a la interfaz `ResultSet`, y contiene el resultado de esta consulta. Los métodos de `ResultSet` permiten al programa manipular el resultado de la consulta.

En las líneas 47 a 61 se procesa el objeto `ResultSet`. En la línea 47 se obtienen los *metadatos* para el objeto `ResultSet` y se asignan a una referencia `ResultSetMetaData` (paquete `java.sql`). Los metadatos describen el contenido del objeto `ResultSet`. Los programas pueden usar metadatos mediante la programación, para obtener información acerca de los nombres y tipos de las columnas del objeto `ResultSet`. En la línea 48 se utiliza el método `getColumnCount` de `ResultSetMetaData` para recuperar el número de columnas en el objeto `ResultSet`. En las líneas 50 y 51 se anexan los nombres de las columnas al objeto `StringBuffer` llamado `resultados`.

Observación de ingeniería de software 23.7

Los metadatos permiten a los programas procesar el contenido de un objeto `ResultSet` *en forma dinámica, cuando no se conoce de antemano la información detallada acerca del objeto* `ResultSet`.

En las líneas 55 a 61 se anexan los datos en cada fila del objeto `ResultSet` al objeto `StringBuffer` llamado `resultados`. Antes de procesar el objeto `ResultSet`, el programa posiciona el *cursor* del objeto `ResultSet` en la primera fila de este objeto, mediante el método `next` de `ResultSet` (línea 55). El cursor apunta a la fila actual. El método `next` devuelve el valor `boolean` `true` si el cursor puede posicionarse en la siguiente fila; en caso contrario el método devuelve `false`.

Error común de programación 23.8

Inicialmente, un cursor `ResultSet` *se posiciona antes de la primera fila. Si trata de acceder al contenido de un objeto* `ResultSet` *antes de posicionar el cursor* `ResultSet` *en la primera fila con el método* `next`, *se produce una excepción* `SQLException`.

Si hay filas en el objeto `ResultSet`, en la línea 58 se extrae el contenido de la fila actual. Al procesar un objeto `ResultSet`, es posible extraer cada columna de este objeto como un tipo de Java específico. De hecho, el método `getColumnType` de `ResultSetMetaData` devuelve un valor entero constante de la clase `Types` (paquete `java.sql`), indicando el tipo de una columna especificada. Los programas pueden utilizar estos valores en una estructura `switch` para invocar métodos de `ResultSet` que devuelvan los valores de las columnas como tipos de Java apropiados. Si el tipo de una columna es `Types.INT`, el método `getInt` de `ResultSet` devuelve el valor de la columna como un `int`. Los métodos *get* de `ResultSet` generalmente reciben como argumento un *número de columna* (como un valor `int`) o un *nombre de columna* (como un valor `String`), indicando cuál valor de la columna se va a obtener. Visite la página

```
java.sun.com/j2se/1.4.1/docs/guide/jdbc/getstart/
   GettingStartedTOC.fm.html
```

para obtener las asociaciones detalladas de los tipos SQL y los tipos de Java, y para determinar el método de `ResultSet` apropiado a llamar a cada tipo de SQL.

Tip de rendimiento 23.2

Si una consulta especifica las columnas exactas a seleccionar de la base de datos, entonces el objeto `ResultSet` *contendrá las columnas en el orden especificado. En este caso, es más eficiente utilizar el número de columna para obtener su valor que utilizar su nombre. El número de columna proporciona un acceso directo a la columna especificada. Si se usa el nombre, se requiere una búsqueda lineal de los nombres de columna para localizar la apropiada.*

Por cuestiones de simpleza, en este ejemplo trataremos a cada valor como un objeto `Object`. El programa recupera el valor de cada columna con el método `getObject` de `ResultSet` (línea 58) y anexa la representación `String` del objeto `Object` a `resultados`. Observe que, a diferencia de los índices de arreglos que empiezan en 0, los números de columna de `ResultSet` empiezan en 1.

Error común de programación 23.9

Si se especifica el número de columna 0 cuando se van a obtener valores de un objeto `ResultSet`, *se produce una excepción* `SQLException`.

Error común de programación 23.10

Al tratar de manipular un objeto `ResultSet` *después de cerrar el objeto* `Statement` *que lo creó, se produce una excepción* `SQLException`. *El programa descarta el objeto* `ResultSet` *cuando se cierra el objeto* `Statement` *correspondiente.*

Observación de ingeniería de software 23.8

Cada objeto `Statement` *puede abrir solamente un objeto* `ResultSet` *en un momento dado. Cuando un objeto* `Statement` *devuelve un nuevo objeto* `ResultSet`, *el objeto* `Statement` *cierra el objeto* `ResultSet` *anterior. Para utilizar varios objetos* `ResultSet` *en paralelo, se deben usar objetos* `Statement` *separados para devolver los objetos* `ResultSet`.

En las líneas 64 a 70 se crea la GUI que muestra el objeto `StringBuffer` llamado `resultados`, establece el tamaño de la ventana de aplicación y muestra esta ventana en pantalla. El bloque `finally` (líneas 91 a 106) cierra el objeto `Statement` (línea 94) y el objeto `Connection` que mantiene la conexión a la base de datos (línea 95).

Para ejecutar éste y los demás ejemplos del capítulo, primero debe ejecutar el archivo de procesamiento por lotes `setCP.bat` (o la secuencia de comandos de shell `setCP.ksh` para UNIX) que viene en el directorio `Cloudscape_5.1\frameworks\embedded\bin`, para establecer la ruta de clases apropiada.

23.6.2 Consultas en la base de datos `libros`

El siguiente ejemplo (figuras 23.27 y 23.30) mejora el ejemplo de la figura 23.25 al permitir al usuario introducir cualquier consulta en el programa. Este ejemplo muestra el resultado de una consulta en un objeto `JTable`, utilizando un objeto `TableModel` para proporcionar los datos del objeto `ResultSet` al objeto `JTable`. La clase `ResultSetTableModel` (figura 23.27) realiza la conexión a la base de datos y mantiene el objeto `ResultSet`. La clase `MostrarResultadosConsulta` (figura 23.30) crea la GUI y especifica una instancia de la clase `ResultSetTableModel` para proporcionar datos para el objeto `JTable`.

http://librosysolucionarios.net

La clase **ResultSetTableModel** (figura 23.27) extiende a la clase ***AbstractTableModel*** (paquete **javax.swing.table**), la cual implementa a la interfaz **TableModel**. La clase **ResultSetTableModel** redefine a los métodos **getColumnClass**, **getColumnCount**, **getColumnName**, **getRowCount** y **getValueAt** de **TableModel**. Las implementaciones predeterminadas de los métodos **isCellEditable** y **setValueAt** de **TableModel** (proporcionados por **AbstractTableModel**) no se redefinen, ya que este ejemplo no soporta la capacidad de editar las celdas del objeto **JTable**. Además, tampoco se redefinen las implementaciones predeterminadas de los métodos **addTableModelListener** y **removeTableModelListener** de **TableModel** (proporcionados por **AbstractTableModel**), ya que las implementaciones de estos métodos de **AbstractTableModel** agregan y eliminan apropiadamente los componentes de escucha de eventos.

```
1  // Fig. 23.27: ResultSetTableModel.java
2  // Un objeto TableModel que proporciona los datos de un objeto ResultSet a un objeto JTable.
3
4  import java.sql.*;
5  import java.util.*;
6  import javax.swing.table.*;
7
8  // Las filas y columnas del objeto ResultSet se cuentan desde 1 y las filas
9  // y columnas del objeto JTable se cuentan desde 0. Al procesar filas
10 // o columnas del objeto ResultSet para usarlas en un objeto JTable, es
11 // necesario sumar 1 al número de fila o columnas para manipular la
12 // columna apropiada del objeto ResultSet (es decir, la columna 0 del objeto JTable
13 // es la columna 1 del objeto ResultSet y la fila 0 del objeto JTable es la fila 1 del
   objeto ResultSet).
14 public class ResultSetTableModel extends AbstractTableModel {
15    private Connection conexion;
16    private Statement instruccion;
17    private ResultSet conjuntoResultados;
18    private ResultSetMetaData metaDatos;
19    private int numeroDeFilas;
20
21    // mantener el registro del estado de la conexión a la base de datos
22    private boolean conectadoALaBaseDeDatos = false;
23
24    // inicializar conjuntoResultados y obtener su objeto de meta datos;
25    // determinar el número de filas
26    public ResultSetTableModel( String controlador, String url,
27       String consulta ) throws SQLException, ClassNotFoundException
28    {
29       // cargar clase de controlador de base de datos
30       Class.forName( controlador );
31
32       // conectarse a la base de datos
33       conexion = DriverManager.getConnection( url );
34
35       // crear objeto Statement para consultar la base de datos
36       instruccion = conexion.createStatement(
37          ResultSet.TYPE_SCROLL_INSENSITIVE,
38          ResultSet.CONCUR_READ_ONLY );
39
40       // actualizar estado de conexión a la base de datos
41       conectadoALaBaseDeDatos = true;
```

Figura 23.27 **ResultSetTableModel** permite a un objeto **JTable** mostrar el contenido de un objeto **ResultSet**. (Parte 1 de 4.)


```
42
43       // establecer consulta y ejecutarla
44       establecerConsulta( consulta );
45    }
46
47    // obtener la clase que representa al tipo de columna
48    public Class getColumnClass( int columna ) throws IllegalStateException
49    {
50       // asegurar que la conexión a la base de datos esté disponible
51       if ( !conectadoALaBaseDeDatos )
52          throw new IllegalStateException( "No hay conexion a la base de datos" );
53
54       // determinar la clase de Java de columna
55       try {
56          String nombreClase = metaDatos.getColumnClassName( columna + 1 );
57
58          // devolver objeto Class que representa a nombreClase
59          return Class.forName( nombreClase );
60       }
61
62       // atrapar excepciones SQLException y ClassNotFoundException
63       catch ( Exception excepcion ) {
64          excepcion.printStackTrace();
65       }
66
67       // si ocurren problemas arriba, suponer que es tipo Object
68       return Object.class;
69    }
70
71    // obtener el número de columnas en el objeto ResultSet
72    public int getColumnCount() throws IllegalStateException
73    {
74       // asegurar que la conexión a la base de datos esté disponible
75       if ( !conectadoALaBaseDeDatos )
76          throw new IllegalStateException( "No hay conexion a la base de datos" );
77
78       // determinar el número de columnas
79       try {
80          return metaDatos.getColumnCount();
81       }
82
83       // atrapar excepciones SQLException e imprimir mensaje de error
84       catch ( SQLException excepcionSQL ) {
85          excepcionSQL.printStackTrace();
86       }
87
88       // si ocurren problemas arriba, devolver 0 para el número de columnas
89       return 0;
90    }
91
92    // obtener el nombre de una columna específica en el objeto ResultSet
93    public String getColumnName( int columna ) throws IllegalStateException
94    {
95       // asegurar que la conexión a la base de datos esté disponible
```

Figura 23.27 **ResultSetTableModel** permite a un objeto **JTable** mostrar el contenido de un objeto **ResultSet**. (Parte 2 de 4.)

http://librosysolucionarios.net

```
96       if ( !conectadoALaBaseDeDatos )
97          throw new IllegalStateException( "No hay conexion a la base de datos" );
98
99       // determinar el nombre de la columna
100       try {
101          return metaDatos.getColumnName( columna + 1 );
102       }
103
104       // atrapar excepciones SQLException e imprimir mensaje de error
105       catch ( SQLException excepcionSQL ) {
106          excepcionSQL.printStackTrace();
107       }
108
109       // si hay problemas, devolver cadena vacía para el nombre de la columna
110       return "";
111    }
112
113    // devolver el número de filas en el objeto ResultSet
114    public int getRowCount() throws IllegalStateException
115    {
116       // asegurar que la conexión a la base de datos esté disponible
117       if ( !conectadoALaBaseDeDatos )
118          throw new IllegalStateException( "No hay conexion a la base de datos" );
119
120       return numeroDeFilas;
121    }
122
123    // obtener el valor en una fila y columna específicas
124    public Object getValueAt( int fila, int columna )
125       throws IllegalStateException
126    {
127       // asegurar que la conexión a la base de datos esté disponible
128       if ( !conectadoALaBaseDeDatos )
129          throw new IllegalStateException( "No hay conexion a la base de datos" );
130
131       // obtener un valor en una fila y columna específicas del objeto ResultSet
132       try {
133          conjuntoResultados.absolute( fila + 1 );
134
135          return conjuntoResultados.getObject( columna + 1 );
136       }
137
138       // atrapar excepciones SQLException e imprimir mensaje de error
139       catch ( SQLException excepcionSQL ) {
140          excepcionSQL.printStackTrace();
141       }
142
143       // si hay problemas, devolver objeto cadena vacía
144       return "";
145    }
146
147    // establecer nueva cadena de consulta para la base de datos
148    public void establecerConsulta( String consulta )
149       throws SQLException, IllegalStateException
```

Figura 23.27 `ResultSetTableModel` permite a un objeto `JTable` mostrar el contenido de un objeto `ResultSet`. (Parte 3 de 4.)

http://librosysolucionario.net


```
150    {
151       // asegurar que la conexión a la base de datos esté disponible
152       if ( !conectadoALaBaseDeDatos )
153          throw new IllegalStateException( "No hay conexion a la base de datos" );
154
155       // especificar consulta y ejecutarla
156       conjuntoResultados = instruccion.executeQuery( consulta );
157
158       // obtener meta datos para el objeto ResultSet
159       metaDatos = conjuntoResultados.getMetaData();
160
161       // determinar el número de filas en el objeto ResultSet
162       conjuntoResultados.last();                    // mover a la última fila
163       numeroDeFilas = conjuntoResultados.getRow();  // obtener número de fila
164
165       // notificar al objeto JTable que el modelo ha cambiado
166       fireTableStructureChanged();
167    }
168
169    // cerrar objetos Statement y Connection
170    public void desconectarDeLaBaseDeDatos()
171    {
172       // cerrar objetos Statement y Connection
173       try {
174          instruccion.close();
175          conexion.close();
176       }
177
178       // atrapar excepciones SQLException e imprimir mensaje de error
179       catch ( SQLException excepcionSQL ) {
180          excepcionSQL.printStackTrace();
181       }
182
183       // actualizar estado de conexión a la base de datos
184       finally {
185          conectadoALaBaseDeDatos = false;
186       }
187    }
188
189 } // fin de la clase ResultSetTableModel
```

Figura 23.27 `ResultSetTableModel` permite a un objeto `JTable` mostrar el contenido de un objeto `ResultSet`. (Parte 4 de 4.)

El constructor de `ResultSetTableModel` (líneas 26 a 45) recibe tres argumentos `String`: el nombre de la clase del controlador, el URL de la base de datos y la consulta predeterminada a realizar. El constructor lanza cualquier excepción que ocurra en su cuerpo, de vuelta a la aplicación que creó el objeto `ResultSetTableModel`, para que la aplicación pueda determinar cómo manejar esa excepción (por ejemplo, reportar un error y terminar la aplicación). En la línea 30 se carga el controlador de la base de datos. En la línea 33 se establece una conexión a la base de datos. En las líneas 36 a 38 se invoca el método `createStatement` de `Connection` para obtener un objeto `Statement`. En este ejemplo utilizamos una versión del método `createStatement` que toma dos argumentos: el *tipo de conjunto de resultados* y la *concurrencia del conjunto de resultados*. El tipo de conjunto de resultados (figura 23.28) especifica si el cursor del objeto `ResultSet` puede desplazarse en ambas direcciones o solamente hacia delante, y si el objeto `ResultSet` es sensible a los cambios. Los objetos `ResultSet` que son susceptibles a los cambios los reflejan inmediatamente después de que éstos se realizan con los métodos de la interfaz `ResultSet`. Si un objeto

Constante de tipo static de ResultSet	Descripción
`TYPE_FORWARD_ONLY`	Especifica que el cursor del objeto `ResultSet` puede desplazarse solamente en dirección hacia delante (es decir, desde la primera hasta la última fila en el objeto `ResultSet`).
`TYPE_SCROLL_INSENSITIVE`	Especifica que el cursor del objeto `ResultSet` puede desplazarse en cualquier dirección y que los cambios realizados al objeto `ResultSet` durante su procesamiento no se reflejarán en este objeto, a menos que el programa consulte la base de datos otra vez.
`TYPE_SCROLL_SENSITIVE`	Especifica que el cursor del objeto `ResultSet` puede desplazarse en cualquier dirección y que los cambios realizados al objeto `ResultSet` durante su procesamiento se reflejarán inmediatamente en este objeto.

Figura 23.28 Constantes de `ResultSet` para especificar el tipo del objeto `ResultSet`.

Constante de concurrencia static de ResultSet	Descripción
`CONCUR_READ_ONLY`	Especifica que un objeto `ResultSet` no puede actualizarse (es decir, los cambios en el contenido del objeto `ResultSet` no pueden reflejarse en la base de datos con los métodos *update* de `ResultSet`).
`CONCUR_UPDATABLE`	Especifica que un objeto `ResultSet` puede actualizarse (es decir, los cambios en el contenido del objeto `ResultSet` pueden reflejarse en la base de datos con los métodos *update* de `ResultSet`).

Figura 23.29 Constantes de `ResultSet` para especificar las propiedades de los resultados.

`ResultSet` no es susceptible a los cambios, la consulta que produjo ese objeto `ResultSet` debe ejecutarse de nuevo para reflejar cualquier cambio realizado. La concurrencia del conjunto de resultados (figura 23.29) especifica si el objeto `ResultSet` puede actualizarse con los métodos de actualización de `ResultSet`. En este ejemplo utilizamos un objeto `ResultSet` que puede desplazarse, que no es susceptible a los cambios y es de sólo lectura. En la línea 44 se invoca el método `establecerConsulta` de `ResultSetTableModel` (líneas 148 a 167) para realizar la consulta predeterminada.



Tip de portabilidad 23.5

Algunos controladores JDBC no soportan objetos `ResultSet` desplazables. En dichos casos, generalmente el controlador devuelve un objeto `ResultSet` en el que el cursor puede moverse sólo hacia delante. Para obtener más información, consulte la documentación de su controlador de bases de datos.



Tip de portabilidad 23.6

Algunos controladores JDBC no soportan objetos `ResultSet` actualizables. En dichos casos, generalmente el controlador devuelve un objeto `ResultSet` de sólo lectura. Para obtener más información, consulte la documentación de su controlador de bases de datos.



Error común de programación 23.11

Al intentar actualizar un objeto `ResultSet` cuando el controlador de base de datos no soporta objetos `ResultSet` actualizables se producen excepciones `SQLException`.



Error común de programación 23.12

Al intentar mover el cursor hacia atrás mediante un objeto `ResultSet`, cuando el controlador de base de datos no soporta el desplazamiento hacia atrás, se produce una excepción `SQLException`.

http://librosysolucionarios.net

El método `getColumnClass` (líneas 48 a 69) devuelve un objeto `Class` que representa a la superclase de todos los objetos en una columna específica. El objeto `JTable` utiliza esta información para configurar el desplegador de celdas y el editor de celdas predeterminados para esa columna en el objeto `JTable`. En la línea 56 se utiliza el método *`getColumnClassName`* de `ResultSetMetaData` para obtener el nombre de clase completamente calificado para la columna especificada. En la línea 59 se carga la clase y se devuelve el objeto `Class` correspondiente. Si ocurre una excepción, el bloque `catch` en las líneas 63 a 65 imprime un rastreo de la pila y en la línea 68 se devuelve `Object.class` (la instancia de `Class` que representa a la clase `Object`) como el tipo predeterminado. (*Nota:* En la línea 56 se utiliza el argumento `columna + 1`. Al igual que los arreglos, los números de fila y columna del objeto `JTable` se cuentan desde 0. Sin embargo, los números de fila y columna del objeto `ResultSet` se cuentan desde 1. Por lo tanto, al procesar filas o columnas de un objeto `ResultSet` para utilizarlas en un objeto `JTable`, es necesario sumar 1 al número de fila o de columna para manipular la fila o columna apropiada del objeto `ResultSet`.)

El método `getColumnCount` (líneas 72 a 90) devuelve el número de columnas en el objeto `ResultSet` subyacente del modelo. En la línea 80 se utiliza el método *`getColumnCount`* de `ResultSetMetaData` para obtener el número de columnas en el objeto `ResultSet`. Si ocurre una excepción, el bloque `catch` en las líneas 84 a 86 imprime un rastreo de la pila y en la línea 89 se devuelve 0 como el número predeterminado de columnas.

El método `getColumnName` (líneas 93 a 111) devuelve el nombre de la columna en el objeto `ResultSet` subyacente del modelo. En la línea 101 se utiliza el método *`getColumnName`* de `ResultSetMetaData` para obtener el nombre de la columna del objeto `ResultSet`. Si ocurre una excepción, el bloque `catch` en las líneas 105 a 107 imprime un rastreo de la pila y en la línea 110 se devuelve la cadena vacía como el nombre predeterminado de la columna.

El método `getRowCount` (líneas 114 a 121) devuelve el número de filas en el objeto `ResultSet` subyacente del modelo. Cuando el método `establecerConsulta` (líneas 148 a 167) realiza una consulta, almacena el número de filas en la variable `numeroDeFilas`.

El método `getValueAt` (líneas 124 a 145) devuelve el objeto `Object` en una fila y columna específicas del objeto `ResultSet` subyacente del modelo. En la línea 133 se utiliza el método *`absolute`* de `ResultSet` para posicionar el cursor del objeto `ResultSet` en una fila específica. En la línea 135 se utiliza el método `getObject` de `ResultSet` para obtener el objeto `Object` en una columna específica de la fila actual. Si ocurre una excepción, el bloque `catch` en las líneas 139 a 141 imprime un rastreo de la pila y en la línea 144 se devuelve el objeto cadena vacío como el valor predeterminado.

El método `establecerConsulta` (líneas 148 a 167) ejecuta la consulta que recibe como argumento para obtener un nuevo objeto `ResultSet` (línea 156). En la línea 159 se obtiene el objeto `ResultSetMetaData` para el nuevo objeto `ResultSet`. En la línea 162 se utiliza el método *`last`* de `ResultSet` para posicionar el cursor de `ResultSet` en la última fila del objeto `ResultSet`. En la línea 163 se utiliza el método *`getRow`* de `ResultSet` para obtener el número de fila de la fila actual en el objeto `ResultSet`. En la línea 166 se invoca el método *`fireTableStructureChanged`* (heredado de la clase `AbstractTableModel`) para notificar a cualquier objeto `JTable` que utilice a este objeto `ResultSetTableModel` como su modelo, que la estructura del modelo ha cambiado. Esto hace que el objeto `JTable` vuelva a llenar sus filas y columnas con los datos del nuevo objeto `ResultSet`. El método `establecerConsulta` lanza cualquier excepción que ocurra en su cuerpo, devuelta a la aplicación que invocó a `establecerConsulta`.

El método `desconectarDeLaBaseDeDatos` (líneas 170 a 187) implementa un método de terminación apropiado para la clase `ResultSetTableModel`. Recuerde que en el capítulo 8 vimos que no se recomienda utilizar el método `finalize` de `Object` para "limpiar" los restos que dejan los objetos, ya que no hay garantía de que el método `finalize` vaya a ser llamado. En vez de ello, un diseñador de clases debe proporcionar un método público que los clientes de la clase deban invocar en forma explícita para liberar los recursos que haya utilizado un objeto. En este caso, el método `desconectarDeLaBaseDeDatos` cierra los objetos `instruccion` y `conexion` de la base de datos (líneas 174 y 175), los cuales se consideran recursos limitados. Los clientes de la clase `ResultSetTableModel` deben siempre invocar a este método cuando la instancia de esta clase ya no se necesite. Además, observe que cada uno de los otros métodos en la clase lanzarán una excepción `IllegalStateException` si el campo booleano `conectadoALaBaseDeDatos` es `false`. El método `desconectarDeLaBaseDeDatos` establece a `conectadoALaBaseDeDatos` en

false (línea 185) para asegurarse que los clientes no utilicen una instancia de **ResultSetTableModel** después de que ésta haya sido eliminada. **IllegalStateExcepcion** es una excepción de las bibliotecas de Java que es apropiada para indicar esta condición de error.

El constructor de **MostrarResultadosConsulta** (figura 23.30, líneas 25 a 159) crea un objeto **ResultSetTableModel** y la GUI para la aplicación. En las líneas 15, 16 y 19 se declaran el nombre de la clase del controlador de base de datos, el URL de la base de datos y la consulta predeterminada que se pasan al constructor de **ResultSetTableModel** para realizar la conexión inicial a la base de datos y ejecutar la consulta predeterminada. En la línea 58 se crea el objeto **JTable** y se pasa un objeto **ResultSetTableModel** al constructor de **JTable**, el cual a su vez registra el objeto **JTable** como un componente de escucha para los eventos **TableModelEvent** que sean generados por el objeto **ResultSetTableModel**. En las líneas 66 a 110 se registra un manejador de eventos para el **botonEnviar** en el que el usuario hace clic para enviar una consulta a la base de datos. Cuando el usuario hace clic en el botón, el método **actionPerformed** (líneas 71 a 106) invoca al método **establecerConsulta** de **ResultSetTableModel** para ejecutar la nueva consulta. Si la consulta del usuario falla (por ejemplo, debido a un error de sintaxis en la entrada del usuario), en las líneas 87 y 88 se ejecuta la consulta predeterminada. Si la consulta predeterminada también falla, podría haber un error más grave, por lo que en la línea 98 se asegura que se cierre la conexión a la base de datos y en la línea 100 se sale del programa. Las capturas de pantalla de la figura 23.30 muestran los resultados de dos consultas. La primera pantalla muestra la consulta predeterminada, en la que se recuperan todos los datos de la tabla **autores** de la base de datos **libros**. La segunda pantalla muestra una consulta que selecciona el nombre de pila y el apellido paterno de cada autor en la tabla **autores**, y combina esa información con el título y número de edición de la tabla **titulos**. Pruebe a introducir sus propias consultas en el área de texto y haga clic en el botón **Enviar consulta** para enviar la consulta.

```
1  // Fig. 23.30: MostrarResultadosConsulta.java
2  // Mostrar el contenido de la tabla Autores en la
3  // base de datos libros.
4
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.sql.*;
8  import java.util.*;
9  import javax.swing.*;
10 import javax.swing.table.*;
11
12 public class MostrarResultadosConsulta extends JFrame {
13
14    // controlador JDBC y URL de la base de datos
15    static final String CONTROLADOR_JDBC = "com.ibm.db2j.jdbc.DB2jDriver";
16    static final String URL_BASEDEDATOS = "jdbc:db2j:libros";
17
18    // la consulta predeterminada recupera todos los datos de la tabla autores
19    static final String CONSULTA_PREDETERMINADA = "SELECT * FROM autores";
20
21    private ResultSetTableModel modeloTabla;
22    private JTextArea areaConsulta;
23
24    // crear objeto ResultSetTableModel y la GUI
25    public MostrarResultadosConsulta()
26    {
27       super( "Mostrando resultados de la consulta" );
28
29       // crear objeto ResultSetTableModel y mostrar tabla de la base de datos
```

Figura 23.30 **MostrarResultadosConsulta** para consultar la base de datos **libros**. (Parte 1 de 4.)


```
30       try {
31
32          // especificar la ubicación de la base de datos en el sistema de archivos
33          System.setProperty( "db2j.system.home", "C:/Cloudscape_5.1" );
34
35          // crear objeto TableModel para los resultados de la consulta SELECT * FROM autores
36          modeloTabla = new ResultSetTableModel( CONTROLADOR_JDBC, URL_BASEDEDATOS,
37             CONSULTA_PREDETERMINADA );
38
39          // establecer objeto JTextArea en el que el usuario escribe las consultas
40          areaConsulta = new JTextArea( CONSULTA_PREDETERMINADA, 3, 100 );
41          areaConsulta.setWrapStyleWord( true );
42          areaConsulta.setLineWrap( true );
43
44          JScrollPane panelDesplazable = new JScrollPane( areaConsulta,
45             ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
46             ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
47
48          // establecer objeto JButton para enviar las consultas
49          JButton botonEnviar = new JButton( "Enviar consulta" );
50
51          // crear objeto Box para manejar la colocación de areaConsulta y
52          // botonEnviar en la GUI
53          Box cuadro = Box.createHorizontalBox();
54          cuadro.add( panelDesplazable );
55          cuadro.add( botonEnviar );
56
57          // crear delegado de JTable para modeloTabla
58          JTable tablaResultados = new JTable( modeloTabla );
59
60          // colocar componentes de la GUI en el panel de contenido
61          Container c = getContentPane();
62          c.add( cuadro, BorderLayout.NORTH );
63          c.add( new JScrollPane( tablaResultados ), BorderLayout.CENTER );
64
65          // crear componente de escucha de eventos para botonEnviar
66          botonEnviar.addActionListener(
67
68             new ActionListener() {
69
70                // pasar consulta al modelo de la tabla
71             public void actionPerformed( ActionEvent evento )
72             {
73                // realizar una nueva consulta
74                try {
75                   modeloTabla.establecerConsulta( areaConsulta.getText() );
76                }
77
78                // atrapar posibles excepciones SQLException al realizar una nueva
                     consulta
79                catch ( SQLException excepcionSQL ) {
80                   JOptionPane.showMessageDialog( null,
81                      excepcionSQL.getMessage(), "Error en la base de datos",
82                      JOptionPane.ERROR_MESSAGE );
```

Figura 23.30 **MostrarResultadosConsulta** para consultar la base de datos **libros**. (Parte 2 de 4.)

http://librosysoluciones.net

```
83
84                // tratar de recuperarse de una consulta inválida del usuario
85                // ejecutando la consulta predeterminada
86                try {
87                   modeloTabla.establecerConsulta( CONSULTA_PREDETERMINADA );
88                   areaConsulta.setText( CONSULTA_PREDETERMINADA );
89                }
90
91                // atrapar posible excepcion SQLException al realizar
                     consulta predeterminada
92                   catch ( SQLException excepcionSQL2 ) {
93                      JOptionPane.showMessageDialog( null,
94                       excepcionSQL2.getMessage(), "Error en la base de datos",
95                      JOptionPane.ERROR_MESSAGE );
96
97                   // asegurar que la conexión a la base de datos esté cerrada
98                   modeloTabla.desconectarDeLaBaseDeDatos();
99
100                  System.exit( 1 );   // terminar la aplicación
101
102                  } // fin de bloque catch interno
103
104               } // fin de bloque catch externo
105
106            } // fin de actionPerformed
107
108         } // fin de la clase interna ActionListener
109
110      }; // fin de la llamada a addActionListener
111
112      // establecer tamaño de ventana y mostrarla en pantalla
113      setSize( 500, 250 );
114      setVisible( true );
115
116   } // fin de bloque try
117
118   // atrapar la excepción ClassNotFoundException lanzada por el
119   // objeto ResultSetTableModel si no se encuentra el controlador de la base de datos
120      catch ( ClassNotFoundException claseNoEncontrada ) {
121         JOptionPane.showMessageDialog( null,
122         "No se encontro el controlador de Cloudscape", "No se encontro
            el controlador",
123         JOptionPane.ERROR_MESSAGE );
124
125         System.exit( 1 );   // terminar la aplicación
126      } // fin de bloque catch
127
128   // atrapar la excepción SQLException lanzada por el objeto ResultSetTableModel
129   // si ocurren problemas al establecer la conexión a la base de datos
130   // y realizar la consulta en la misma
131   catch ( SQLException excepcionSQL ) {
132      JOptionPane.showMessageDialog( null, excepcionSQL.getMessage(),
133      "Error en la base de datos", JOptionPane.ERROR_MESSAGE );
134
```

Figura 23.30 `MostrarResultadosConsulta` para consultar la base de datos `libros`. (Parte 3 de 4.)


```
135      // asegurar que la conexión a la base de datos esté cerrada
136      modeloTabla.desconectarDeLaBaseDeDatos();
137
138      System.exit( 1 );   // terminar la aplicación
139   }
140
141   // desechar la ventana cuando el usuario salga de la aplicación
142   // (esta opción sobrescribe a la opción predeterminada de HIDE_ON_CLOSE)
143   setDefaultCloseOperation( DISPOSE_ON_CLOSE );
144
145   // asegurar que la conexión a la base de datos esté cerrada cuando el usuario
        salga de la aplicación
146   addWindowListener(
147
148      new WindowAdapter() {
149
150         // desconectarse de la base de datos y salir cuando se haya cerrado la ventana
151         public void windowClosed( WindowEvent evento )
152         {
153            modeloTabla.desconectarDeLaBaseDeDatos();
154            System.exit( 0 );
155         }
156      }
157   );
158
159} // fin del constructor de MostrarResultadosConsulta
160
161// ejecutar la aplicación
162public static void main( String args[] )
163{
164   new MostrarResultadosConsulta();
165}
166
167} // fin de la clase MostrarResultadosConsulta
```

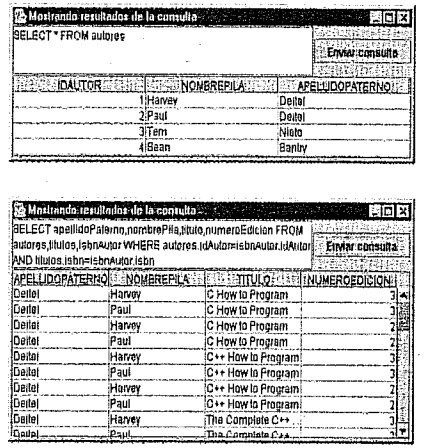


Figura 23.30 `MostrarResultadosConsulta` para consultar la base de datos `libros`. (Parte 4 de 4.)

http://librosysoluciones.net

23.7 Procedimientos almacenados

Muchos sistemas de administración de bases de datos pueden almacenar instrucciones de SQL individuales o conjuntos de instrucciones de SQL en una base de datos, para que los programas que tengan acceso a esa base de datos puedan invocar esas instrucciones. A dichas instrucciones de SQL se les conoce como *procedimientos almacenados*. JDBC permite a los programas invocar procedimientos almacenados mediante el uso de objetos que implementen a la interfaz `CallableStatement`. Los objetos `CallableStatement` pueden recibir argumentos especificados con los métodos heredados de la interfaz `PreparedStatement`. Además, los objetos `CallableStatement` pueden especificar *parámetros de salida*, en los cuales un procedimiento almacenado puede colocar valores de retorno. La interfaz `CallableStatement` incluye métodos para especificar cuáles parámetros en un procedimiento almacenado son parámetros de salida. La interfaz también incluye métodos para obtener los valores de los parámetros de salida devueltos de un procedimiento almacenado.



Tip de portabilidad 23.7

Aunque la sintaxis para crear procedimientos almacenados difiere entre los diversos sistemas de administración de bases de datos, la interfaz `CallableStatement` proporciona una interfaz uniforme para especificar los parámetros de entrada y salida de los procedimientos almacenados, así como para invocarlos.



Tip de portabilidad 23.8

De acuerdo con la documentación de la API para la interfaz `CallableStatement`, para lograr una máxima portabilidad entre los sistemas de bases de datos, los programas deben procesar las cuentas de actualización o los objetos `ResultSet` devueltos de un objeto `CallableStatement` antes de obtener los valores de cualquier parámetro de salida.

23.8 Recursos en Internet y World Wide Web

`java.sun.com/products/jdbc/`
La página inicial sobre JDBC de Sun Microsystems, Inc.

`java.sun.com/docs/books/tutorial/jdbc/index.html`
La trayectoria del tutorial de Java sobre JDBC de Sun Microsystems, Inc.

`www.sql.org`
Este portal de SQL proporciona vínculos hacia muchos recursos, incluyendo la sintaxis de SQL, tips, tutoriales, libros, revistas, grupos de discusión, compañías con servicios de SQL, consultores de SQL y software gratuito.

`industry.java.sun.com/products/jdbc/drivers`
El motor de búsqueda de Sun Microsystems, Inc. para localizar controladores de JDBC.

`java.sun.com/j2se/1.3/docs/guide/jdbc/index.html`
La documentación de la API JDBC de Sun Microsystems, Inc.

`java.sun.com/products/jdbc/faq.html`
Las preguntas frecuentes acerca de JDBC de Sun Microsystems, Inc.

`www.jguru.com/jguru/faq/faqpage.jsp?name=JDBC`
Las FAQs sobre JDBC de JGuru.

`http://www-306.ibm.com/software/data/cloudscape/`
Desde este sitio podrá descargar la versión más reciente de Cloudscape y acceder a toda su documentación en línea.

`java.sun.com/products/jdbc/articles/package2.html`
Una descripción general sobre la API del paquete opcional JDBC 2.0.

`http://java.sun.com/j2se/1.4.1/docs/api/javax/sql/RowSet.html`
Acceso anticipado a las implementaciones de `RowSet` de Sun. [*Nota:* Tal vez necesite registrarse en Java Developer Connection (`developer.java.sun.com/developer/index.html`) para poder descargar software de este sitio.]

`developer.java.sun.com/developer/Books/JDBCTutorial/chapter5.html`
El capítulo 5 (tutorial de `RowSet`) del libro *The JDBC 2.0 API Tutorial and Reference, Second Edition.*

RESUMEN

- Una base de datos es una colección integrada de datos. Un sistema de administración de bases de datos (DBMS) proporciona mecanismos para almacenar y organizar datos.
- Los sistemas de administración de bases de datos más populares hoy en día son los sistemas de bases de datos relacionales.
- SQL es un lenguaje utilizado casi universalmente con sistemas de bases de datos relacionales, para realizar consultas y manipular datos.
- Los programas se conectan a (e interactúan con) bases de datos relacionales a través de una interfaz: software que facilita las comunicaciones entre un sistema de administración de bases de datos y un programa.
- Los programadores en Java se comunican con las bases de datos y manipulan sus datos utilizando la API JDBC. Un controlador de JDBC implementa la interfaz para una base de datos específica.
- Una base de datos relacional almacena los datos en tablas. Las tablas están compuestas de filas, y las filas están compuestas de columnas en las que se almacenan los valores.
- Una clave principal proporciona un valor único que no puede duplicarse en otras filas.
- Cada columna de la tabla representa un atributo distinto.
- La clave principal puede estar compuesta por más de una columna.
- SQL proporciona un conjunto completo de instrucciones que permiten a los programadores definir consultas complejas para recuperar datos de una base de datos.
- Toda columna en una clave principal debe tener un valor, y el valor de la clave principal debe ser único. A esto se le conoce como la Regla de integridad de entidades.
- Una relación de uno a varios entre tablas indica que una fila en una tabla puede tener muchas filas en otra tabla.
- Una clave externa debe concordar con el valor de la clave principal en otra tabla.
- La clave externa ayuda a mantener la Regla de la integridad referencial. Cada valor de la columna de clave externa debe aparecer en la columna de clave principal de otra tabla. Las claves externas permiten que se una la información de varias tablas para fines de análisis. Hay una relación de uno a varios entre una clave principal y su correspondiente clave externa.
- La forma básica de una consulta `SELECT` es:

  `SELECT * FROM` *nombreDeTabla*

  en donde el asterisco (`*`) indica que deben seleccionarse todas las columnas de *nombreDeTabla* y *nombreDeTabla* especifica la tabla en la base de datos de la cual se van a recuperar los datos.
- Para recuperar ciertas columnas de una tabla, reemplace el asterisco (`*`) con una lista separada por comas de los nombres de las columnas.
- Los programadores procesan los resultados de una consulta conociendo de antemano el orden de las columnas en el resultado. Al especificar las columnas explícitamente, se garantiza que siempre se devuelvan en el orden especificado, incluso aunque el orden real en la(s) tabla(s) sea distinto.
- La cláusula `WHERE` opcional en una consulta `SELECT` especifica los criterios de selección para la consulta. La forma básica de una consulta `SELECT` con criterios de selección es:

  `SELECT` *nombreDeColumna1, nombreDeColumna2, ...* `FROM` *nombreDeTabla* `WHERE` *criterios*
- La cláusula `WHERE` puede contener los operadores `<`, `>`, `<=`, `>=`, `=`, `<>` y `LIKE`. El operador `LIKE` se utiliza para relacionar patrones con los caracteres comodines por ciento (`%`) y guión bajo (`_`).
- Un carácter de por ciento (`%`) en un patrón indica que una cadena que concuerde con ese patrón puede tener cero o más caracteres en la ubicación del carácter de por ciento en el patrón.
- Un guión bajo (`_`) en la cadena del patrón indica un carácter individual en esa posición en el patrón.
- El resultado de una consulta puede ordenarse en forma ascendente o descendente, mediante el uso de la cláusula `ORDER BY` opcional. La forma más simple de una cláusula `ORDER BY` es:

  `SELECT` *nombreDeColumna1, nombreDeColumna2, ...* `FROM` *nombreDeTabla* `ORDER BY` *columna* `ASC`
  `SELECT` *nombreDeColumna1, nombreDeColumna2, ...* `FROM` *nombreDeTabla* `ORDER BY` *columna*
  `DESC`

  en donde `ASC` especifica el orden ascendente, `DESC` especifica el orden descendente y *columna* especifica la columna en la cual se basa el ordenamiento. El orden predeterminado es ascendente, por lo que `ASC` es opcional.

http://librosysolucionarios.net

- Pueden usarse varias columnas para ordenar mediante una cláusula `ORDER BY`, de la siguiente forma:

  `ORDER BY` *columna1 tipoDeOrden, columna2 tipoDeOrden, ...*

- Las cláusulas `WHERE` y `ORDER BY` pueden combinarse en una consulta. Si se utiliza, la cláusula `ORDER BY` debe ser la última cláusula en la consulta.
- Un operador `INNER JOIN` fusiona las filas de dos tablas al relacionar los valores en una columna que sea común para las dos tablas. La forma básica del operador `INNER JOIN` es:

  `SELECT` *nombreDeColumna1, nombreDeColumna2, ...*
  `FROM` *tabla1*
  `INNER JOIN` *tabla2*
  `ON` *tabla1.nombreDeColumna = tabla2.nombreDeColumna*

  La cláusula `ON` especifica las columnas de cada tabla que se van a comparar para determinar cuáles filas se van a unir. Si una instrucción de SQL utiliza columnas con el mismo nombre provenientes de varias tablas, los nombres de las columnas deben estar completamente calificados con su nombre de tabla y un operador punto (.).
- Una instrucción `INSERT` inserta una nueva fila en una tabla. La forma básica de esta instrucción es:

  `INSERT INTO` *nombreDeTabla (nombreDeColumna1, nombreDeColumna2, ..., nombreDeColumnaN)*
  `VALUES` *(valor1, valor2, ..., valorN)*

  en donde *nombreDeTabla* es la tabla en la que se va a insertar la fila. El *nombreDeTabla* va seguido de una lista separada por comas de los nombres de columnas, entre paréntesis. La lista de nombres de columnas va seguida por la palabra clave `VALUES` de SQL, y una lista separada por comas de valores entre paréntesis.
- Las instrucciones de SQL utilizan el carácter de comilla sencilla (') como delimitador para las cadenas. Para especificar una cadena que contenga una comilla sencilla en una instrucción de SQL, la comilla sencilla debe escaparse con otra comilla sencilla.
- Una instrucción `UPDATE` modifica los datos en una tabla. La forma básica de la instrucción `UPDATE` es:

  `UPDATE` *nombreDeTabla*
  `SET` *nombreDeColumna1 = valor1, nombreDeColumna2 = valor2, ..., nombreDeColumnaN = valorN*
  `WHERE` *criterios*

  en donde *nombreDeTabla* es la tabla de la que se van a actualizar los datos. El *nombreDeTabla* va seguido por la palabra clave `SET` y una lista separada por comas de los pares nombre/valor de las columnas, en el formato *nombreDeColumna=valor*. Los *criterios* de la cláusula `WHERE` determinan las filas que se van a actualizar.
- Una instrucción `DELETE` elimina filas de una tabla. La forma más simple de una instrucción `DELETE` es:

  `DELETE FROM` *nombreDeTabla* `WHERE` *criterios*

en donde *nombreDeTabla* es la tabla de la que se va a eliminar una fila (o filas). Los *criterios* de `WHERE` determinan cuál(es) fila(s) se va(n) a eliminar.

- El paquete `java.sql` contiene clases e interfaces para manipular bases de datos relacionales en Java.
- Un programa debe cargar la clase del controlador de bases de datos para que pueda conectarse a la base de datos.
- JDBC soporta cuatro categorías de controladores: el controlador puente JDBC a ODBC (tipo 1); el controlador nativo de API desarrollado parcialmente en Java (tipo 2); el controlador JDBC-Net desarrollado puramente en Java (tipo 3) y el controlador de protocolo nativo desarrollado puramente en Java (tipo 4). Los controladores 3 y 4 son preferidos, ya que son soluciones desarrolladas puramente en Java.
- Un objeto que implementa a la interfaz `Connection` administra la conexión entre el programa de Java y la base de datos. Los objetos `Connection` permiten a los programas crear instrucciones de SQL para manipular los datos y realizar el procesamiento de transacciones.
- El método `getConnection` de la clase `DriverManager` trata de conectarse a una base de datos especificada por su argumento URL. El URL ayuda al programa a localizar la base de datos. El URL incluye el protocolo y el subprotocolo de comunicación, junto con el nombre de la base de datos.
- El método `createStatement` de `Connection` crea un objeto de tipo `Statement`. El programa utiliza al objeto `Statement` para enviar instrucciones de SQL a la base de datos.
- El método `executeQuery` de `Statement` ejecuta una consulta y devuelve un objeto que implementa a la interfaz `ResultSet` que contiene el resultado de la consulta. Los métodos de `ResultSet` permiten a un programa manipular el resultado de la consulta.

- Un objeto `ResultSetMetaData` describe el contenido de un objeto `ResultSet`. Los programas pueden usar metadatos mediante la programación, para obtener información acerca de los nombres y tipos de las columnas del objeto `ResultSet`.
- El método `getColumnCount` de `ResultSetMetaData` recupera el número de columnas en el objeto `ResultSet`.
- El método `next` de `ResultSet` posiciona el cursor de `ResultSet` en la siguiente fila del objeto `ResultSet`. El cursor apunta a la fila actual. El método `next` devuelve el valor `boolean true` si el cursor puede posicionarse en la siguiente fila; en caso contrario el método devuelve `false`. Este método debe llamarse para empezar a procesar un objeto `ResultSet`.
- Al procesar objetos `ResultSet`, es posible extraer cada columna del objeto `ResultSet` como un tipo de Java específico. El método `getColumnType` de `ResultSetMetaData` devuelve un valor entero constante de la clase `Types` (paquete `java.sql`), indicando el tipo de los datos de una columna específica.
- Los métodos *get* de `ResultSet` generalmente reciben como argumento un número de columna (como un valor `int`) o un nombre de columna (como un valor `String`), indicando cuál valor de la columna se va a obtener.
- Los números de fila y columna de un objeto `ResultSet` empiezan en 1.
- Cada objeto `Statement` puede abrir solamente un objeto `ResultSet` en un momento dado. Cuando un objeto `Statement` devuelve un nuevo objeto `ResultSet`, el objeto `Statement` cierra el objeto `ResultSet` anterior.
- El método `createStatement` de `Connection` tiene una versión sobrecargada que toma dos argumentos: el tipo y la concurrencia del resultado. El tipo del resultado especifica si el cursor del objeto `ResultSet` puede desplazarse en ambas direcciones o solamente hacia delante, y si el objeto `ResultSet` es susceptible a los cambios. La concurrencia del resultado especifica si el objeto `ResultSet` puede actualizarse con los métodos de actualización de `ResultSet`.
- Algunos controladores JDBC no soportan objetos `ResultSet` desplazables y/o actualizables.
- El método `getColumnClass` de `TableModel` devuelve un objeto `Class` que representa a la superclase de todos los objetos en una columna específica. El objeto `JTable` utiliza esta información para configurar el desplegador de celdas y el editor de celdas predeterminados para esa columna en un objeto `JTable`.
- El método `getColumnClassName` de `ResultSetMetaData` obtiene el nombre completamente calificado de la columna especificada.
- El método `getColumnCount` de `TableModel` devuelve el número de columnas en el objeto `ResultSet` subyacente del modelo.
- El método `getColumnName` de `TableModel` devuelve el nombre de la columna en el objeto `ResultSet` subyacente del modelo.
- El método `getColumnName` de `ResultSetMetaData` obtiene el nombre de la columna proveniente del objeto `ResultSet`.
- El método `getRowCount` de `TableModel` devuelve el número de filas en el objeto `ResultSet` subyacente del modelo.
- El método `getValueAt` de `TableModel` devuelve el objeto `Object` en una fila y columna específicas del objeto `ResultSet` subyacente del modelo.
- El método `absolute` de `ResultSet` posiciona el cursor de `ResultSet` en una fila específica.
- El método `fireTableStructureChanged` de `AbstractTableModel` notifica a cualquier objeto `JTable` que utilice un objeto `TableModel` específico como su modelo, que los datos en el modelo han cambiado.
- JDBC permite a los programas invocar procedimientos almacenados mediante el uso de objetos que implementen a la interfaz `CallableStatement`.
- Los objetos `CallableStatement` pueden recibir argumentos especificados con los métodos heredados de la interfaz `PreparedStatement`. Además, los objetos `CallableStatement` pueden especificar parámetros de salida, en los cuales un procedimiento almacenado puede colocar valores de retorno.

TERMINOLOGÍA

`%`, carácter comodín de SQL
`_`, carácter comodín de SQL
`absolute`, método de `ResultSet`
`AbstractTableModel`, clase
`addTableModelListener`, método de `TableModel`
base de datos
base de datos relacional
`CallableStatement`, interfaz
clave externa
clave principal
`close`, método de `Connection`
`close`, método de `Statement`
Cloudscape, base de datos

http://librosysolucionarios.net

columna
`com.ibm.db2j.jdbc.DB2jDriver`
conectarse a una base de datos
`Connection`, interfaz
consultar una base de datos
controlador de base de datos
controlador JDBC
controlador tipo 1 (puente JDBC a ODBC)
controlador tipo 2 (nativo de API, desarrollado parcialmente en Java)
controlador tipo 3 (JDBC-Net desarrollado puramente en Java)
controlador tipo 4 (protocolo nativo, desarrollado puramente en Java)
`createStatement`, método de `Connection`
criterios de selección
`DELETE`, instrucción de SQL
`deleteRow`, método de `ResultSet`
`DriverManager`, clase
`execute`, método de `Statement`
`executeQuery`, método de `Statement`
`executeUpdate`, método de `Statement`
fila
`fireTableStructureChanged`, método de `AbstractTableModel`
`getColumnClass`, método de `TableModel`
`getColumnClassName`, método de `ResultSetMetaData`
`getColumnCount`, método de `ResultSetMetaData`
`getColumnCount`, método de `TableModel`
`getColumnName`, método de `ResultSetMetaData`
`getColumnName`, método de `TableModel`
`getColumnType`, método de `ResultSetMetaData`
`getConnection`, método de `DriverManager`
`getMetaData`, método de `ResultSet`
`getMoreResults`, método de `Statement`
`getObject`, método de `ResultSet`
`getResultSet`, método de `Statement`
`getRow`, método de `ResultSet`
`getRowCount`, método de `TableModel`
`getUpdateCount`, método de `Statement`
`getValueAt`, método de `TableModel`
`INNER JOIN`, operador de SQL
`INSERT`, instrucción de SQL
`insertRow`, método de `ResultSet`
`java.sql`, paquete
`javax.swing.table`, paquete
JDBC
JdbcOdbcDriver
`last`, método de `ResultSet`
metadatos
`moveToCurrentRow`, método de `ResultSet`
`moveToInsertRow`, método de `ResultSet`
`next`, método de `ResultSet`
`ORDER BY`, cláusula de una instrucción de SQL
parámetro de salida
procedimiento almacenado
Regla de integridad de entidades
Regla de integridad referencial
relación de uno a varios
relacionar patrones
`removeTableModelListener`, método de `TableModel`
resultado
`ResultSet` actualizable
`ResultSet`, interfaz
`ResultSet`, tipos
`ResultSetMetaData`, interfaz
secuencia de comandos de SQL
`SELECT`, instrucción de SQL
`setString`, método de `PreparedStatement`
SQL
`SQLException`, clase
`Statement`, interfaz
tabla
`TableModel`, interfaz
`TableModelEvent`, clase
`Types`, clase
unir
`UPDATE`, instrucción de SQL
`updateRow`, método de `ResultSet`
`WHERE`, cláusula de una instrucción de SQL

EJERCICIOS DE AUTOEVALUACIÓN

23.1 Complete las siguientes oraciones:

a) El lenguaje de consulta de bases de datos estándar en la industria es _____.
b) Una tabla en una base de datos consiste de _____ y _____.
c) Los resultados de una consulta de SQL se manipulan en Java como objetos _____.
d) La _____ identifica en forma única a cada fila en una tabla.
e) La palabra clave _____ de SQL va seguida por los criterios de selección que especifican las filas a seleccionar en una consulta.
f) Las palabras clave _____ de SQL especifican la manera en que se ordenan las filas en una consulta.
g) Al proceso de fusionar filas de varias tablas de una base de datos se le conoce como _____ las tablas.
h) Una _____ es una colección organizada de datos.
i) Una _____ es un conjunto de columnas cuyos valores concuerdan con los valores de clave principal de otra tabla.
j) El paquete _____ contiene clases e interfaces para manipular bases de datos relacionales en Java.
k) La interfaz _____ ayuda a administrar la conexión entre un programa de Java y una base de datos.
l) Un objeto _____ se utiliza para enviar una consulta a una base de datos.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

23.1 a) SQL. b) Filas, columnas. c) `ResultSet`. d) Clave principal. e) `WHERE`. f) `ORDER BY`. g) Unir. h) Base de datos. i) Clave externa. j) `java.sql`. k) `Connection`. l) `Statement`.

EJERCICIOS

23.2 Utilizando las técnicas mostradas en este capítulo, defina una aplicación de consulta completa para la base de datos `libros`. Proporcione una serie de consultas predefinidas, con un nombre apropiado para cada consulta, mostrada en un objeto `JComboBox`. Además, permita a los usuarios suministrar sus propias consultas y agregarlas al objeto `JcomboBox`. Proporcione las siguientes consultas predefinidas:

a) Seleccionar todos los autores de la tabla `autores`.
b) Seleccionar todas las editoriales de la tabla `editoriales`.
c) Seleccionar un autor específico y mostrar todos los libros de ese autor. Incluir el título, año e ISBN. Ordenar la información alfabéticamente, en base al apellido paterno y nombre de pila del autor.
d) Seleccionar una editorial específica y mostrar todos los libros publicados por esa editorial. Incluir el título, año e ISBN. Ordenar la información alfabéticamente por título.
e) Proporcione cualquier otra consulta que usted crea que sea apropiada.

23.3 Modifique el ejercicio 23.2 para definir una aplicación de manipulación de bases de datos completa para la base de datos `libros`. Además de las consultas, el usuario deberá poder editar los datos existentes y agregar nuevos datos a la base de datos (obedeciendo las restricciones de integridad referencial y de entidades). Permita al usuario editar la base de datos de las siguientes formas:

a) Agregar un nuevo autor.
b) Editar la información existente para un autor.
c) Agregar un nuevo título para un autor. (Recuerde que el libro debe tener una entrada en la tabla `isbnAutor`.) Asegúrese de especificar la editorial de ese título.
d) Agregar una nueva editorial.
e) Editar la información existente para una editorial.
f) Para cada una de las manipulaciones anteriores de bases de datos, diseñe una GUI apropiada para permitir al usuario realizar la manipulación de los datos.

23.4 En la sección 10.7 presentamos una jerarquía nómina-empleado para calcular la nómina de cada empleado. En este ejercicio proporcionamos una base de datos de empleados que corresponde a la jerarquía nómina-empleado. (En el CD que acompaña a este libro se proporciona una secuencia de comandos de SQL para crear la base de datos de empleados, junto con los ejemplos de este capítulo, y también en nuestro sitio Web `www.deitel.com`.) Escriba una aplicación que permita:

a) Agregar empleados a la tabla Empleado.
b) Para cada empleado agregado a la tabla, agregar la nómina a la tabla correspondiente. Por ejemplo, para un empleado asalariado agregue la información de nómina a la tabla empleadosAsalariados.

23.5 Escriba una aplicación que proporcione un objeto `JComboBox` y un objeto `JTextArea` para permitir al usuario realizar una consulta que se seleccione del objeto `JComboBox` o que se defina en el objeto `JTextArea`. Las consultas predefinidas de ejemplo son:

a) Seleccionar a todos los empleados que trabajen en el departamento de VENTAS.
b) Seleccionar a los empleados por horas que trabajen más de 30 horas.
c) Seleccionar a todos los empleados por comisión en orden descendente en base a la tasa de comisión.

23.6 Modifique el ejercicio 23.5 para realizar las siguientes tareas:

a) Incrementar el salario base en un 10% para todos los empleados base más comisión.
b) Si el cumpleaños del empleado es en el mes actual, agregar un bono de $100.
c) Para todos los empleados por comisión cuyas ventas brutas sean mayores de 10000, agregar un bono de $100.

BIBLIOGRAFÍA

Ashmore, D. C. "Best Practices for JDBC Programming". *Java Developers Journal*, 5: núm. 4 (2000): 42-54.
Blaha, M. R., W. J. Premerlani y J. E. Rumbaugh. "Relational Database Design Using an Object-Oriented Methodology." *Communications of the ACM*, 31: núm. 4 (1988): 414-427.
Bowler, R. J. "The Evolution of Connecting." *Java Developers Journal*, 5: núm. 10 (2000): 24-26.
Brunner, R. J. "After the Connection." *Java Developers Journal*, 5: núm. 11 (2000): 42-46.

http://librosysolucionarios.net

Callahan, T. "So You Want a Stand-Alone Database for Java." *Java Developers Journal*, 3: núm. 12 (1998): 28-36.
Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, junio de 1970.
Codd, E. F. "Further Normalization of the Data Base Relational Model." *Courant Computer Science Symposia*, Vol. 6, *Data Base Systems*. Upper Saddle River, NJ: Prentice Hall, 1972.
Codd, E. F. "Fatal Flaws in SQL." *Datamation*, 34: núm. 16 (1988): 45-48.
Cooper, J. W. "Making Databases Easier for Your Users." *Java Pro*, 4: núm. 10 (2000): 47-54.
Date, C. J. *An Introduction to Database Systems, Seventh Edition*. Reading, MA: Addison Wesley, 2000.
Deitel, H. M. *Operating Systems, Second Edition*. Reading, MA: Addison Wesley, 1990.
Duguay, C. "Electronic Mail Merge." *Java Pro*, Winter 1999/2000, 22-32.
Ergul, S. "Transaction Processing with Java." *Java Report*, enero de 2001, 30-36.
Fisher, M. "JDBC Database Access," (una pista en *The Java Tutorial*), `<java.sun.com/docs/books/tutorial/jdbc/index.html>`.
Harrison, G., "Browsing the JDBC API," *Java Developers Journal*, 3: núm. 2 (1998): 44-52.
Jasnowski, M. "Persistence Frameworks," *Java Developers Journal*, 5: núm. 11 (2000): 82-86.
"JDBC API Documentation," `<java.sun.com/j2se/1.3/docs/guide/jdbc/index.html>`.
Jordan, D. "An Overview of Sun's Java Data Objects Specification," *Java Pro*, 4: núm. 6 (2000): 102-108.
Khanna, P. "Managing Object Persistence with JDBC," *Java Pro*, 4: núm. 5 (2000): 28-33.
Reese, G. *Database Programming with JDBC and Java, Second Edition*. Cambridge, MA: O'Reilly, 2001.
Spell, B. "Create Enterprise Applications with JDBC 2.0," *Java Pro*, 4: núm. 4 (2000): 40-44.
Stonebraker, M. "Operating System Support for Database Management," *Communications of the ACM* , 24: núm. 7 (1981): 412-418.
Taylor, A. *JDBC Developer's Resource: Database Programming on the Internet*. Upper Saddle River, NJ: Prentice Hall, 1999.
Thilmany, C. "Applying Patterns to JDBC Development," *Java Developers Journal*, 5: núm. 6 (2000): 80-90.
Venugopal, S. 2000. "Cross-Database Portability with JDBC, *Java Developers Journal*, 5: núm. 1 (2000): 58-62.
White, S., M. Fisher, R. Cattell, G. Hamilton y M. Hapner. *JDBC API Tutorial and Reference, Second Edition*. Boston, MA: Addison Wesley, 1999.
Winston, A. "A Distributed Database Primer," *UNIX World*, abril de 1988, 54-63.

24

Servlets

Objetivos

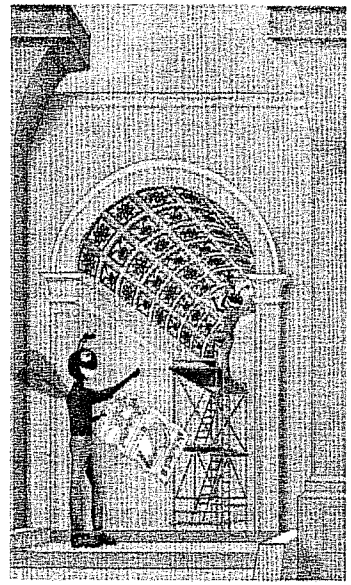
- Ejecutar servlets con el servidor Apache Tomcat.
- Responder a las peticiones de HTTP de un objeto `HttpServlet`.
- Redireccionar las peticiones a recursos Web estáticos y dinámicos.

Una petición justa debe ir seguida por el acto en silencio.
Dante Alighieri

Se dice que la parte más larga del viaje es pasar la puerta.
Marcus Terentius Varro

Si soy nominado, no aceptaré; si soy electo, no serviré.
General William T. Sherman

Los amigos comparten todo.
Pitágoras



http://librosysolucionarios.net

Plan general

24.1 Introducción
24.2 Generalidades y arquitectura de los servlets
24.2.1 La interfaz `Servlet` y el ciclo de vida de un servlet
24.2.2 La clase `HttpServlet`
24.2.3 La interfaz `HttpServletRequest`
24.2.4 La interfaz `HttpServletResponse`
24.3 Manejo de las peticiones `get` de HTTP
24.3.1 Configuración del servidor Apache Tomcat
24.3.2 Despliegue de una aplicación Web
24.4 Manejo de peticiones `get` de HTTP que contienen datos
24.5 Manejo de las peticiones `post` de HTTP
24.6 Redirección de peticiones a otros recursos
24.7 Aplicaciones multinivel: Uso de JDBC desde un servlet
24.8 Recursos en Internet y World Wide Web
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

24.1 Introducción

Hay mucho entusiasmo en cuanto a Internet y World Wide Web. Internet enlaza el "mundo de la información". World Wide Web facilita el uso de Internet y le brinda las herramientas multimedia. Las organizaciones ven a Internet y a la Web como algo imprescindible para las estrategias de sus sistemas de información. Java proporciona una variedad de herramientas de red integradas que facilitan el desarrollo de aplicaciones basadas en Internet y Web. Java no solamente puede especificar el paralelismo a través del subprocesamiento múltiple, sino que también puede permitir a los programas buscar información en todo el mundo y colaborar con programas ejecutándose en otros equipos a nivel internacional, nacional o sólo dentro de una organización. Java puede incluso permitir a los applets (subprogramas) y aplicaciones que se ejecutan en el mismo equipo comunicarse unos con otros, teniendo en cuenta las restricciones de seguridad.

Las redes son un tema masivo y complejo. Generalmente, los estudiantes de ciencias computacionales e ingeniería computacional toman un curso de nivel superior, con duración de un semestre completo, sobre redes computacionales y continúan estudiando este tema a nivel de graduados. Java proporciona un extenso complemento de herramientas de red que se utilizan como vehículos de implementación en cursos de redes computacionales y en la industria.

Las herramientas de red de Java se agrupan en varios paquetes. Las herramientas de red fundamentales se definen mediante las clases e interfaces del paquete `java.net`. Este paquete ofrece *comunicaciones basadas en sockets*, las cuales permiten a las aplicaciones ver las redes como flujos de datos; un programa puede leer de un *socket* o escribir a un socket tan fácil como leer de un archivo o escribir a un archivo. Las clases e interfaces del paquete `java.net` también ofrecen *comunicaciones basadas en paquetes*, las cuales permiten la transmisión de *paquetes* de información individuales; éstos se utilizan comúnmente para transmitir audio y vídeo a través de Internet. En el capítulo 18 mostramos cómo crear y manipular sockets, y cómo comunicarnos con los paquetes de datos.

Las vistas de niveles más altos en cuanto a las redes son proporcionadas por las clases e interfaces de los paquetes `java.rmi` (cinco paquetes) para *Invocación de métodos remotos (RMI)* y los paquetes `org.omg` (siete paquetes) para la *Arquitectura común de agente de solicitud de objetos (CORBA)* que forman parte de la API Java 2. Los paquetes de RMI permiten a objetos de Java ejecutarse en máquinas virtuales de Java separadas (normalmente en equipos separados) para comunicarse a través de llamadas a métodos remotos. Dichas llamadas parecen invocar métodos en un objeto del mismo programa, pero en realidad tienen integrados los

servicios de red (con base en las herramientas del paquete `java.net`), con lo cual las llamadas a los métodos se comunican hacia otro objeto en un equipo separado. Los paquetes de CORBA proporcionan una funcionalidad similar a los paquetes de RMI. Una diferencia clave entre RMI y CORBA es que RMI puede utilizarse solamente entre objetos de Java, mientras que CORBA puede usarse entre dos aplicaciones cualesquiera que entiendan CORBA (incluyendo aplicaciones escritas en otros lenguajes de programación).[1] En el capítulo 13 de nuestro libro *Advanced Java 2 Platform How to Program*, presentamos las herramientas de RMI de Java; y en los capítulos 26 y 27 del mismo libro hablamos sobre los conceptos básicos de CORBA y presentamos un ejemplo práctico en el que se implementa un sistema distribuido en CORBA.

Nuestra discusión sobre las redes se enfoca en ambos lados de una *relación cliente-servidor*. El *cliente* solicita la realización de cierta acción, mientras que el *servidor* realiza la acción y responde al cliente. Este modelo de comunicación tipo petición-respuesta es la base de las vistas de nivel superior sobre las redes en Java: *servlets* y *JavaServer Pages (JSP)*. Un servlet extiende la funcionalidad de un servidor, como un servidor Web. Los paquetes `javax.servlet` y `javax.servlet.http` proporcionan las clases e interfaces para definir a los servlets. Los paquetes `javax.servlet.jsp` y `javax.servlet.jsp.tagext` proporcionan las clases e interfaces que extienden las capacidades de los servlets para usar JavaServer Pages. Mediante el uso de una sintaxis especial, JSP permite a los implementadores de páginas Web crear páginas que utilicen una funcionalidad encapsulada de Java, e incluso les permite escribir *scriptlets* de verdadero código de Java directamente en la página.

Una implementación común del modelo de petición-respuesta se da entre los navegadores y los servidores de World Wide Web. Cuando un usuario selecciona un sitio Web para navegar a través de su navegador (la aplicación cliente), se envía una petición al servidor Web apropiado (la aplicación servidor). Generalmente el servidor responde al cliente enviando la página Web de XHTML apropiada.[2] Los servlets son efectivos para desarrollar soluciones basadas en Web que ayuden a proporcionar un acceso seguro a un sitio Web, a interactuar con bases de datos en beneficio de un cliente, a generar en forma dinámica documentos de XHTML para que los navegadores los muestren en pantalla, y a mantener información de sesión única para cada cliente.

En este capítulo empezaremos nuestras discusiones sobre las redes, con servlets que mejoran la funcionalidad de los servidores World Wide Web: la forma más común de servlet hoy en día. En el capítulo 25 hablaremos sobre las JSPs, que se traducen para convertirse en servlets. Las JSPs son una manera conveniente y poderosa de implementar el mecanismo de petición/respuesta de la Web, sin necesidad de entrar en los detalles de nivel bajo de los servlets. En conjunto, los servlets y las JSPs forman el nivel Web de Java 2 Enterprise Edition (J2EE).

Muchos desarrolladores sienten que los servlets son la solución correcta para las aplicaciones que utilizan bases de datos en forma intensiva, las cuales se comunican con los denominados *clientes delgados* (aplicaciones que requieren de un soporte mínimo del lado cliente). El servidor es responsable del acceso a la base de datos. Los clientes se conectan al servidor utilizando los protocolos estándar disponibles en la mayoría de las plataformas cliente. Por lo tanto, el código de la lógica de presentación para generar contenido dinámico puede escribirse una vez y residir en el servidor para que los clientes lo utilicen, lo cual permite a los programadores crear clientes delgados eficientes.

En este capítulo, nuestros ejemplos de servlets demuestran el mecanismo de petición/respuesta Web (principalmente con peticiones `get` y `post`), redireccionando las peticiones hacia otros recursos e interactuando con bases de datos a través de JDBC. Colocamos este capítulo después de nuestra discusión sobre JDBC y las bases de datos intencionalmente, para poder crear aplicaciones multinivel cliente-servidor que utilicen bases de datos.

Sun Microsystems, a través del *Proceso de la comunidad Java (Java Community Process)*, es responsable del desarrollo de las especificaciones para los servlets y JavaServer Pages. La implementación de referencia de ambos estándares se encuentra bajo desarrollo por la *Fundación de software Apache* (`www.apache.org`), como parte del *Proyecto Jakarta* (`jakarta.apache.org`). Como se menciona en la página inicial del

1. La *Invocación de métodos remotos sobre el Protocolo Inter-Orb de Internet (RMI-IIOP)* permite la integración de Java con objetos distribuidos que no son de Java, mediante el uso de CORBA IIOP.
2. Si usted no está familiarizado con XHTML y CSS, consulte los documentos PDF *Introduction to XHTML* y *Cascading Style Sheets*™ en el CD que acompaña a este libro. Estos documentos también están disponibles en nuestro sitio Web, `www.deitel.com`.

http://librosysolucionarios.net

Proyecto Jakarta, "El objetivo del Proyecto Jakarta es proporcionar soluciones de servidores con calidad comercial, basadas en la Plataforma Java, que se desarrollen en forma abierta y cooperativa". Hay muchos subproyectos bajo el proyecto Jakarta para ayudar a los desarrolladores comerciales del lado servidor. A la parte relacionada con servlets y JSP del Proyecto Jakarta se le conoce como *Tomcat*. Ésta es la implementación de referencia oficial de los estándares de JSP y servlets. En este capítulo utilizaremos a Tomcat para demostrar el funcionamiento de los servlets. La implementación más reciente de Tomcat al momento de escribir este libro era la versión 4.1.12. Para su conveniencia, Tomcat 4.1.12 se incluye en el CD que acompaña a este libro. Sin embargo, la versión más reciente de Tomcat podrá descargarla siempre del sitio Web del Grupo Apache. Para ejecutar los servlets en este capítulo, debe instalar Tomcat o una implementación equivalente de servlets y JavaServer Pages. Hablaremos sobre cómo instalar y configurar Tomcat en las secciones 24.3.1 y 24.3.2, después de presentar nuestro primer ejemplo.

En las instrucciones para probar cada uno de los ejemplos en este capítulo, le indicaremos que debe copiar archivos en ciertos directorios específicos de Tomcat. Todos los archivos de ejemplo para este capítulo se encuentran en el CD que acompaña a este libro y en nuestro sitio Web `www.deitel.com`.

[*Nota:* Al final de la sección 24.8 proporcionamos una lista de especificaciones de Internet (según lo descrito en la Especificación Servlet 2.2) para las tecnologías relacionadas con el desarrollo de servlets. Cada tecnología se enlista con su número RFC (Request for Comments —Petición de comentarios). Le proporcionaremos el URL de un sitio Web que le permita localizar cada especificación para que la analice.]

24.2 Generalidades y arquitectura de los servlets

En esta sección veremos las generalidades acerca de la tecnología de los servlets de Java. Hablaremos sobre las clases, métodos y excepciones relacionadas con los servlets, a nivel superior. En las siguientes secciones presentaremos ejemplos en los que crearemos sistemas multinivel cliente-servidor, utilizando las tecnologías de servlets y JDBC.

Internet ofrece muchos protocolos. El protocolo HTTP (*Protocolo de transferencia de hipertexto*), que forma la base de World Wide Web, utiliza *URLs* (*Localizadores uniformes de recursos*) para localizar recursos en Internet. Los URLs comunes representan archivos o directorios, y pueden representar tareas complejas como búsquedas en bases de datos y en Internet. Para obtener más información acerca de los formatos de los URLs, visite la página `www.w3.org/Addressing`. Para obtener más información sobre el protocolo HTTP, visite la página `www.w3.org/Protocols/HTTP`. Para obtener información acerca de una variedad de temas relacionados con World Wide Web, visite el sitio `www.w3.org`.

La tecnología JavaServer Pages es una extensión de la tecnología de los servlets. Generalmente, las JSPs se utilizan cuando la mayoría del contenido que se envía al cliente es texto estático y marcas, y sólo una pequeña porción del contenido se genera en forma dinámica mediante código de Java. Los servlets se utilizan comúnmente cuando una pequeña porción del contenido que se envía al cliente es texto estático o marcas. De hecho, algunos servlets no producen contenido. En vez de ello, realizan una tarea a beneficio del cliente y después invocan a otros servlets o JSPs para que proporcionen una respuesta. Observe que, en la mayoría de los casos, las tecnologías de servlets y JSP son intercambiables. Al servidor que ejecuta un servlet se le conoce como *contenedor de servlets* o *motor de servlets*.

Los servlets y las JavaServer Pages se han vuelto tan populares que ahora se soportan directamente o mediante complementos independientes en la mayoría de los servidores Web y de aplicaciones, incluyendo a Sun ONE Application Server, Internet Information Services (IIS) de Microsoft, el servidor Apache HTTP, el servidor de aplicaciones WebLogic de BEA, el servidor de aplicaciones WebSphere de IBM, el servidor Web Jigsaw del Consorcio World Wide Web, y muchos más.

Los servlets en este capítulo muestran la comunicación entre clientes y servidores a través del protocolo HTTP. Un cliente envía una petición de HTTP al servidor. El contenedor de servlets recibe la petición y la dirige para que sea procesada por el servlet apropiado. El servlet realiza su procesamiento, que puede incluir interactuar con una base de datos u otros componentes del lado servidor como otros servlets, JSPs o Enterprise JavaBeans. El servlet devuelve sus resultados al cliente; normalmente lo hace en forma de un documento de HTML, XHTML o XML para mostrarlo en un navegador, pero pueden devolverse también otros formatos, como imágenes y datos binarios.

http://librosysolucionarios.net

24.2.1 La interfaz `Servlet` y el ciclo de vida de un servlet

Desde el punto de vista arquitectónico, todos los servlets deben implementar a la interfaz `Servlet`. Al igual que con los métodos clave para applets, los métodos de la interfaz `Servlet` son invocados por el contenedor de servlets. Esta interfaz define cinco métodos, que se describen en la figura 24.1.



Observación de ingeniería de software 24.1

Todos los servlets deben implementar a la interfaz `Servlet` del paquete `javax.servlet`.

El ciclo de vida de un servlet empieza cuando el contenedor de servlets lo carga en memoria; generalmente, en respuesta a la primera petición que recibe el servlet. Antes de que el servlet pueda encargarse de esa petición, el contenedor de servlets invoca al método `init` de ese servlet. Una vez que `init` termina de ejecutarse, el servlet puede responder a su primera petición. Todas las peticiones son manejadas por un método del servlet llamado `service`, el cual recibe la petición, la procesa y envía una respuesta al cliente. Durante el ciclo de vida de un servlet, el método `service` es llamado una vez por cada petición. Cada nueva petición generalmente resulta en un nuevo subproceso de ejecución (creado por el contenedor de servlets) en el que se ejecuta el método `service`. Cuando el contenedor de servlets termina el servlet, se hace una llamada al método `destroy` de ese servlet para liberar sus recursos.



Tip de rendimiento 24.1

Es más eficiente iniciar un nuevo subproceso para cada petición que iniciar un proceso completamente nuevo, como es el caso en algunas otras tecnologías del lado servidor como CGI. (Nota: Al igual que los servlets, la tecnología Fast CGI elimina la sobrecarga de iniciar un nuevo proceso para cada petición.)

Los paquetes de servlets definen dos clases `abstract` que implementan a la interfaz `Servlet`: la clase `GenericServlet` (del paquete `javax.servlet`) y la clase `HttpServlet` (del paquete `javax.servlet.http`). Estas clases proporcionan implementaciones predeterminadas de todos los métodos de `Servlet`. La mayoría de los servlets extienden a `GenericServlet` o a `HttpServlet`, y redefine a algunos de sus métodos, o a todos.

Método	Descripción
`void init( ServletConfig config )`	El contenedor de servlets llama a este método una vez durante el ciclo de ejecución de un servlet, para inicializarlo. El argumento tipo `ServletConfig` es proporcionado por el contenedor de servlets que ejecuta al servlet.
`ServletConfig getServletConfig()`	Este método devuelve una referencia a un objeto que implementa a la interfaz `ServletConfig`. Este objeto proporciona el acceso a la información de configuración del servlet, como sus parámetros de inicialización y el objeto `ServletContext` del servlet, el cual proporciona al servlet el acceso a su entorno (es decir, al contenedor de servlets en el cual se ejecuta ese servlet).
`String getServletInfo()`	Este método es definido por un programador de servlets para que devuelva una cadena que contiene información sobre un servlet, como el autor y la versión.
`void service( ServletRequest peticion, ServletResponse respuesta )`	El contenedor de servlets llama a este método para responder a la petición que un cliente hace al servlet.
`void destroy()`	Este método de "limpieza" se llama cuando un servlet es terminado por su contenedor de servlets. Los recursos utilizados por el servlet, como un archivo abierto o una conexión abierta a una base de datos, deben desasignarse aquí.

Figura 24.1 Métodos de la interfaz `Servlet` (paquete `javax.servlet`).

Los ejemplos en este capítulo extienden a la clase `HttpServlet`, la cual define capacidades de procesamiento mejoradas para los servlets que extienden la funcionalidad de un servidor Web. El método clave en cada servlet es `service`, el cual recibe tanto a un objeto `ServletRequest` como a un objeto `ServletResponse`. Estos objetos proporcionan acceso a los flujos de entrada y salida que permiten al servlet leer datos del cliente y enviar datos al cliente. Estos flujos pueden estar basados en bytes o basados en caracteres. Si ocurren problemas durante la ejecución de un servlet, se lanzan excepciones `ServletException` o `IOException` para indicar el problema.



Observación de ingeniería de software 24.2

Los servlets pueden implementar la interfaz de seguimiento `SingleThreadModel` para indicar que sólo un subproceso de ejecución puede entrar al método `service` en una instancia específica de un servlet, en un momento dado. Cuando un servlet implementa a `SingleThreadModel`, el contenedor de servlets puede crear varias instancias del servlet para que el servlet pueda manejar peticiones múltiples en paralelo. En este caso, tal vez usted necesite proporcionar un acceso sincronizado a los recursos compartidos utilizados por el método `service`.

24.2.2 La clase `HttpServlet`

Los servlets basados en Web generalmente extienden a la clase `HttpServlet`. La clase `HttpServlet` redefine el método `service` para diferenciar entre las peticiones típicas recibidas de un navegador Web cliente. Los dos *tipos de peticiones de HTTP* más comunes (también conocidos como *métodos de petición*) son `get` y `post`. Una petición `get` *obtiene* (o *recupera*) la información de un servidor. Las peticiones `get` se utilizan comúnmente para recuperar un documento de HTML o una imagen. Una petición `post` *publica* (o *envía*) datos en un servidor. Las peticiones `post` se utilizan comúnmente para enviar información, como la información de autenticación o los datos de un *formulario* que recopila la entrada del usuario, a un servidor.

La clase `HttpServlet` define los métodos `doGet` y `doPost` para responder a las peticiones `get` y `post` de un cliente, respectivamente. Estos métodos se llaman mediante el método `service`, el cual se llama cuando llega una petición al servidor. El método `service` determina primero el tipo de petición y después llama al método apropiado para manejar esa petición. Hay otros tipos de peticiones menos comunes que se encuentran más allá del alcance de este libro. Los métodos de la clase `HttpServlet` que responden a los otros tipos de peticiones se muestran en la figura 24.2. Todos ellos reciben parámetros de tipo `HttpServletRequest` y `HttpServletResponse`, y devuelven valores `void`. Los métodos de la figura 24.2 no se utilizan frecuentemente. Para obtener más información acerca del protocolo HTTP, visite la página:

```
www.w3.org/Protocols
```



Observación de ingeniería de software 24.3

No debe redefinir el método `service` en una subclase de `HttpServlet`. Si lo hace, evitará que el servlet pueda diferenciar los distintos tipos de peticiones.

Los métodos `doGet` y `doPost` reciben como argumentos un objeto `HttpServletRequest` y un objeto `HttpServletResponse`, los cuales permiten la interacción entre el cliente y el servidor. Los métodos de `HttpServletRequest` facilitan el acceso a los datos que se proporcionan como parte de la petición. Los

Método	Descripción
`doDelete`	Se llama en respuesta a una petición `delete` de HTTP. Dicha petición se utiliza normalmente para eliminar un archivo de un servidor. Esta petición tal vez no esté disponible en algunos servidores debido a sus inherentes riesgos de seguridad (por ejemplo, el cliente podría eliminar un archivo importante para la ejecución del servidor o de una aplicación).
`doHead`	Se llama en respuesta a una petición `head` de HTTP. Dicha petición se utiliza normalmente cuando el cliente sólo desea los encabezados de una respuesta, como el tipo de contenido y la longitud del contenido de la respuesta.

Figura 24.2 Otros métodos de la clase `HttpServlet`. (Parte 1 de 2.)

Método	Descripción
`doOptions`	Se llama en respuesta a una petición `options` de HTTP. Este método devuelve información al cliente, indicando las opciones de HTTP soportadas por el servidor, como la versión de HTTP (1.0 o 1.1) y los métodos de petición que soporta el servidor.
`doPut`	Se llama en respuesta a una petición `put` de HTTP. Dicha petición se utiliza normalmente para almacenar un archivo en el servidor. Esta opción tal vez no esté disponible en algunos servidores, debido a sus inherentes riesgos de seguridad (por ejemplo, el cliente podría colocar una aplicación ejecutable en el servidor que, en caso de llegar a ejecutarse, pudiera dañar al servidor; tal vez eliminando archivos críticos u ocupando recursos).
`doTrace`	Se llama en respuesta a una petición `trace` de HTTP. Dicha petición se utiliza normalmente para la depuración. La implementación de este método devuelve automáticamente un documento de HTML al cliente, el cual contiene la información del encabezado de la petición (los datos enviados por el navegador como parte de la petición).

Figura 24.2 Otros métodos de la clase `HttpServlet`. (Parte 2 de 2.)

métodos de `HttpServletResponse` facilitan el devolver los resultados del servlet al cliente Web. Hablaremos sobre las interfaces `HttpServletRequest` y `HttpServletResponse` en las siguientes dos secciones.

24.2.3 La interfaz `HttpServletRequest`

Toda llamada a `doGet` o `doPost` para un objeto `HttpServlet` recibe un objeto que implementa a la interfaz `HttpServletRequest`. El servidor Web que ejecuta el servlet crea un objeto `HttpServletRequest` y lo pasa al método `service` del servlet (que, a su vez, lo pasa a `doGet` o a `doPost`). Este objeto contiene la petición del cliente. Se proporciona una variedad de métodos para permitir al servlet procesar la petición del cliente. Algunos de estos métodos son de la interfaz `ServletRequest`; la interfaz que es extendida por `HttpServletRequest`. En la figura 24.3 se presentan algunos métodos clave utilizados en este capítulo. Puede ver una lista completa de los métodos de `HttpServletRequest` en línea, en:

```
java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/http/
HttpServletRequest.html
```

o puede descargar e instalar Tomcat (que describiremos en la sección 24.3.1) y ver la documentación en su equipo local.

Método	Descripción
`String getParameter( String nombre )`	Obtiene el valor de un parámetro enviado al servlet como parte de una petición `get` o `post`. El argumento `nombre` representa el nombre del parámetro.
`Enumeration getParameterNames()`	Devuelve los nombres de todos los parámetros enviados al servlet como parte de una petición `post`.
`String[] getParameterValues( String nombre )`	Para un parámetro con múltiples valores, este método devuelve un arreglo de cadenas que contiene los valores para un parámetro de servlet especificado.

Figura 24.3 Algunos métodos de la interfaz `HttpServletRequest`. (Parte 1 de 2.)

http://librosysolucionarios.net

Método	Descripción
`Cookie[] getCookies()`	Devuelve un arreglo de objetos `Cookie` almacenados en el cliente por el servidor. Los objetos `Cookie` pueden utilizarse para identificar clientes en forma única con el servidor.
`HttpSession getSession( boolean crear )`	Devuelve un objeto `HttpSession` asociado con la sesión de navegación actual del cliente. Este método puede crear un objeto `HttpSession` (argumento `true`) si no existe uno ya para el cliente. Los objetos `HttpSession` se utilizan en formas similares a los objetos `Cookie` para identificar a los clientes en forma única.

Figura 24.3 Algunos métodos de la interfaz `HttpServletRequest`. (Parte 2 de 2.)

24.2.4 La interfaz `HttpServletResponse`

Toda llamada a `doGet` o `doPost` para un objeto `HttpServlet` recibe un objeto que implementa a la interfaz `HttpServletResponse`. El servidor Web que ejecuta el servlet crea un objeto `HttpServletResponse` y lo pasa al método `service` del servlet (que, a su vez, lo pasa a `doGet` o a `doPost`). Este objeto proporciona una variedad de métodos que permiten al servlet formular la respuesta al cliente. Algunos de estos métodos son de la interfaz *`ServletResponse`*: la interfaz que es extendida por `HttpServletResponse`. En la figura 24.4 se presentan algunos métodos clave utilizados en este capítulo. Puede ver una lista completa de los métodos de `HttpServletResponse` en línea, en:

```
java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/http/
    HttpServletResponse.html
```

o puede descargar e instalar Tomcat (que describiremos en la sección 24.3.1) y ver la documentación en su equipo local.

24.3 Manejo de las peticiones `get` de HTTP

El propósito principal de una petición `get` de HTTP es recuperar el contenido de un URL especificado; generalmente el contenido es un documento de HTML o XHTML (es decir, una página Web). El servlet de la figura

Método	Descripción
`void addCookie( Cookie cookie )`	Se utiliza para agregar un objeto `Cookie` al encabezado de la respuesta al cliente. Para determinar si se van a guardar objetos `Cookie` en el cliente, se comprueban la edad máxima del objeto `Cookie` y si la opción para almacenar objetos `Cookie` está habilitada en el cliente.
`ServletOutputStream getOutputStream()`	Obtiene un flujo de salida basado en bytes para enviar datos binarios al cliente.
`PrintWriter getWriter()`	Obtiene un flujo de salida basado en caracteres para enviar datos de texto al cliente.
`void setContentType( String tipo )`	Especifica el tipo MIME de la respuesta al navegador. El tipo MIME ayuda al navegador a determinar cómo mostrar los datos (o, posiblemente, qué otra aplicación se ejecutará para procesar los datos). Por ejemplo, el tipo MIME `"text/html"` indica que la respuesta es un documento de HTML, por lo que el navegador muestra la página de HTML.

Figura 24.4 Algunos métodos de la interfaz `HttpServletResponse`.

http://librosysolucionarios.net

24.5 y el documento de XHTML de la figura 24.6 demuestran el uso de un servlet que maneja peticiones `get` de HTTP. Cuando el usuario hace clic en el botón `Obtener documento de HTML` (figura 24.6), se envía una petición `get` al servlet `ServletDeBienvenida` (figura 24.5). El servlet responde a la petición generando en forma dinámica un documento de XHTML para el cliente, que muestra en pantalla el mensaje `"¡Bienvenido a los servlets!"`. En la figura 24.5 se muestra el código fuente de `ServletDeBienvenida`. En la figura 24.6 se muestra el documento de XHTML que el cliente carga para tener acceso al servlet, y se muestran las capturas de pantalla de la ventana del navegador cliente antes y después de la interacción con el servlet. (*Nota:* En la sección 24.3.1 veremos cómo instalar y configurar Tomcat para ejecutar este ejemplo.)

```
1  // Fig. 24.5: ServletDeBienvenida.java
2  // Un servlet simple para procesar peticiones get.
3
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.io.*;
7
8  public class ServletDeBienvenida extends HttpServlet {
9
10    // procesar peticiones "get" de los clientes
11    protected void doGet( HttpServletRequest peticion,
12       HttpServletResponse respuesta )
13          throws ServletException, IOException
14    {
15       respuesta.setContentType( "text/html" );
16       PrintWriter salida = respuesta.getWriter();
17
18       // enviar página de XHTML al cliente
19
20       // empezar documento de XHTML
21       salida.println( "<?xml version = \"1.0\"?>" );
22
23       salida.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
24          "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
25          "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
26
27       salida.println( "<html xmlns = \"http://www.w3.org/1999/xhtml\">" );
28
29       // sección de encabezado del documento
30       salida.println( "<head>" );
31       salida.println( "<title>Un ejemplo de servlet simple</title>" );
32       salida.println( "</head>" );
33
34       // sección del cuerpo del documento
35       salida.println( "<body>" );
36       salida.println( "<h1>¡Bienvenido a los servlets!</h1>" );
37       salida.println( "</body>" );
38
39       // fin del documento de XHTML
40       salida.println( "</html>" );
41       salida.close();  // cerrar flujo para completar la página
42    }
43 }
```

Figura 24.5 `ServletDeBienvenida` que responde a una petición `get` simple de HTTP

En las líneas 4 y 5 se importan los paquetes **javax.servlet** y **javax.servlet.http**. En este ejemplo utilizamos varios tipos de datos de estos paquetes.

El paquete **javax.servlet.http** proporciona la superclase **HttpServlet** para los servlets que manejan peticiones **get** y **post** de HTTP. Esta clase implementa a la interfaz **javax.servlet.Servlet** y agrega métodos que soportan peticiones del protocolo HTTP. La clase **ServletDeBienvenida** extiende a **HttpServlet** (línea 8) por esta razón.

La superclase **HttpServlet** proporciona el método ***doGet*** para responder a las peticiones **get**. Su funcionalidad predeterminada es indicar un error "Method not allowed" (Método no permitido). Generalmente, este error se indica en Internet Explorer con una página Web que muestra el mensaje "Esta página no puede mostrarse" y en Netscape Navigator con una página Web que muestra el mensaje "Error: 405". En las líneas 11 a 42 se redefine el método **doGet** para proporcionar un procesamiento personalizado a las peticiones **get**. El método **doGet** recibe dos argumentos: un objeto **HttpServletRequest** y un objeto **HttpServletResponse** (ambos del paquete **javax.servlet.http**). El objeto **HttpServletRequest** representa la petición del cliente, y el objeto **HttpServletResponse** representa la respuesta del servidor al cliente. Si un método **doGet** no puede manejar la petición de un cliente, lanza una excepción del tipo ***javax.servlet.ServletException***. Si **doGet** se encuentra un error durante el procesamiento de los flujos (al leer del cliente o escribir al cliente), lanza una excepción ***java.io.IOException***.

Para mostrar una respuesta a una petición **get**, nuestro servlet crea un documento de XHTML que contiene el texto **"¡Bienvenido a los servlets!"**. El texto del documento de XHTML es la respuesta al cliente. La respuesta se envía al cliente a través del objeto **PrintWriter** que se obtiene del objeto **HttpServletResponse**.

En la línea 15 se utiliza el método ***setContentType*** del objeto **respuesta** para especificar el tipo de contenido de los datos que se van a enviar como respuesta al cliente. Esto permite al navegador cliente comprender y manejar el contenido. El tipo del contenido se conoce también como el tipo *MIME* (*Extensión multipropósito de correo Internet*) de los datos. En este ejemplo, el tipo de contenido es ***text/html*** para indicar al navegador que la respuesta es un documento de XHTML. El navegador sabe que debe leer las marcas de XHTML en el documento, darle formato de acuerdo a esas marcas y mostrarlo en su ventana.

En la línea 16 se utiliza el método ***getWriter*** del objeto **respuesta** para obtener una referencia al objeto **PrintWriter** que permite al servidor enviar contenido al cliente. (*Nota:* Si la respuesta son datos binarios como una imagen, se utiliza el método ***getOutputStream*** para obtener una referencia a un objeto **ServletOutputStream**.)

En las líneas 21 a 40 se crea el documento de XHTML, escribiendo cadenas mediante el método ***println*** del objeto **salida**. Este método imprime un carácter de nueva línea después de su argumento **String**. Al desplegar la página Web, el navegador no utiliza el carácter de nueva línea. En vez de ello, el carácter de nueva línea aparece en el código fuente de XHTML que usted podrá ver si selecciona la opción **Código fuente** del menú **Ver** en Internet Explorer, o la opción **Page Source** del menú **View** en Netscape Navigator. En la línea 41 se cierra el flujo de salida, se vacía el búfer de salida y se envía la información al cliente. Esto confirma la respuesta al cliente.

El documento de XHTML de la figura 24.6 proporciona un **formulario** que invoca el servlet definido en la figura 24.5. El parámetro **action** de **form** (**/cpej5/bienvenido1**) especifica la ruta del URL que invoca al servlet, y el parámetro **method** de **form** indica que el navegador envía una petición **get** al servidor, lo cual produce una llamada al método **doGet** del servlet. El URL especificado como **action** en este ejemplo se describirá con detalle en la sección 24.3.2, después de que le mostremos cómo instalar y configurar el *servidor Apache Tomcat* para ejecutar el servlet de la figura 24.5.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
```

Figura 24.6 Documento de HTML en el que la propiedad **action** del elemento **form** invoca a **ServletDeBienvenida** a través del alias **bienvenida1** especificado en **web.xml**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
5  <!-- Fig. 24.6: ServletDeBienvenida.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Manejo de una petición Get de HTTP</title>
10 </head>
11
12 <body>
13    <form action = "/cpej5/bienvenido1" method = "get">
14
15       <p><label>Haga clic en el botón para invocar el servlet
16          <input type = "submit" value = "Obtener documento de HTML" />
17       </label></p>
18
19    </form>
20 </body>
21 </html>
```

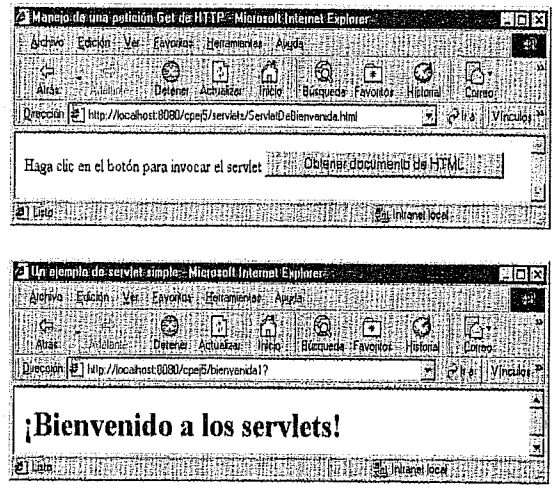


Figura 24.6 Documento de HTML en el que la propiedad **action** del elemento **form** invoca a **ServletDeBienvenida** a través del alias **bienvenida1** especificado en **web.xml**. (Parte 2 de 2.)

Observe que las pantalla de ejemplo muestran a un URL que contiene el nombre de servidor ***localhost*** (un *nombre de host* de servidor muy conocido en la mayoría de las computadoras que soportan protocolos de red basados en TCP/IP como HTTP). A menudo utilizamos **localhost** para demostrar programas de redes en el equipo local, para que los lectores que no tengan una conexión de red puedan aprender de todas formas los conceptos de programación de redes. En este ejemplo, **localhost** indica que el servidor en el que está instalado el servlet se está ejecutando en el equipo local. El nombre de host del servidor va seguido por **:8080**, especificando el número de puerto TCP en el que el servidor Tomcat espera las peticiones de los clientes. Los navegadores Web asumen el puerto TCP 80 de manera predeterminada como el puerto de servidor en el que los clientes hacen peticiones, pero el servidor Tomcat espera las peticiones de los clientes en el puerto TCP 8080. Esto permite a Tomcat ejecutarse en la misma computadora que una aplicación de servidor Web estándar, sin afectar la habilidad de esta aplicación de servidor Web para manejar las peticiones. Si no especificamos explícitamente el número de puerto en el URL, el servlet nunca recibirá nuestra petición y se mostrará un mensaje de error en el navegador.

Observación de ingeniería de software 24.4

La documentación de Tomcat especifica cómo integrarlo con las aplicaciones populares de servidor Web como el servidor Apache HTTP y Microsoft IIS.

Los puertos en este caso no son puertos físicos de hardware a los cuales se conectan cables; en vez de ello, son ubicaciones lógicas identificadas con valores enteros que permiten a los clientes pedir distintos servicios en el mismo servidor. El número de puerto especifica la ubicación lógica en donde un servidor espera y recibe conexiones de los clientes; a éste se le conoce también como el *punto de negociación (handshake point)*. Cuando un cliente se conecta a un servidor para pedir un servicio, el cliente debe especificar el número de puerto para ese servicio; de no ser así, la petición del cliente no puede procesarse. Los números de puerto son enteros positivos con valores hasta 65,535, y hay conjuntos separados de estos números de puerto para los protocolos TCP y UDP. Muchos sistemas operativos reservan los números de puerto menores de 1024 para los servicios del sistema (como los servidores de correo electrónico y World Wide Web). En general, estos puertos no deben especificarse como puertos de conexión en sus propios programas de servidor. De hecho, algunos sistemas operativos requieren de privilegios especiales de acceso para utilizar los números de puerto menores de 1024.

Con tantos puertos para elegir, ¿cómo sabe un cliente cuál puerto utilizar cuando solicita un servicio? El término *número de puerto reconocido* se utiliza a menudo cuando se describen los servicios populares en Internet, como los servidores Web y los servidores de correo electrónico. Por ejemplo, un servidor Web espera que los clientes hagan peticiones en el puerto 80 de manera predeterminada. Todos los navegadores Web conocen este número como el puerto reconocido en un servidor Web, en donde se realizan las peticiones de documentos HTML. Por lo tanto, cuando usted escribe un URL en un navegador Web, éste se conecta normalmente al puerto 80 en el servidor. De manera similar, el servidor Tomcat utiliza el puerto 8080 como su número de puerto. Por lo tanto, las peticiones que se hagan a Tomcat de páginas Web o para invocar servlets y JavaServer Pages deben especificar que el servidor Tomcat espera esas peticiones en el puerto 8080.

El cliente puede acceder al servlet solamente si éste se encuentra instalado en un servidor que pueda responder a las peticiones del servlet. En algunos casos, el soporte para servlets está integrado directamente en el servidor Web, por lo que no se requiere de una configuración especial para manejar las peticiones de servlets. En otros casos es necesario integrar un contenedor de servlets con un servidor Web (como puede hacerse con Tomcat y los servidores Web Apache e IIS). Los servidores Web que soportan servlets generalmente tienen un procedimiento de instalación para los servlets. Si usted planea ejecutar su servlet como parte de un servidor Web, consulte la documentación de su servidor Web para ver cómo instalar un servlet. En nuestros ejemplos demostramos el uso de los servlets con el servidor Apache Tomcat. En la sección 24.3.1 veremos la instalación y configuración de Tomcat para utilizarlo en este capítulo. En la sección 24.3.2 veremos cómo poner en funcionamiento el servlet de la figura 24.5.

24.3.1 Configuración del servidor Apache Tomcat

Tomcat es una implementación completamente funcional de los estándares de JSP y servlets. Incluye un servidor Web, por lo que puede usarse como un contenedor de prueba independiente para JSPs y servlets. Tomcat también puede especificarse como el manejador de las peticiones de JSP y servlets recibidas por servidores Web populares, como el servidor Apache HTTP de la Fundación de software Apache o el servidor Microsoft Internet Information Services (IIS). Tomcat está integrado en la implementación de referencia Java 2 Enterprise Edition de Sun Microsystems.

Puede descargar una versión reciente de Tomcat de:

```
http://jakarta.apache.org/site/binindex.cgi
```

en donde hay una variedad de archivos comprimidos. La implementación completa de Tomcat está contenida en los archivos que empiezan con el nombre `jakarta-tomcat-4.1.12`. Se proporcionan archivos zip, exe, tar y tar comprimido.

Extraiga el contenido del archivo comprimido en un directorio de su disco duro. Para los ejemplos de este libro, utilizamos el directorio `jakarta-tomcat-4.1.12`. Para que Tomcat funcione correctamente, debe definir las variables de entorno `JAVA_HOME` y `CATALINA_HOME`. `JAVA_HOME` debe apuntar al directorio que contenga su instalación de Java (el nuestro es `c:\j2sdk1.4.1`) y `CATALINA_HOME` debe apuntar al directorio que contenga a Tomcat (el nuestro es `c:\jakarta-tomcat-4.1.12`).

Tip para prevenir errores 24.1

En algunas plataformas, tal vez necesite reiniciar su equipo para que las nuevas variables de entorno hagan efecto.

Después de establecer las variables de entorno, podrá iniciar el servidor Tomcat. Abra una ventana de símbolo del sistema (o shell) y cambie el directorio actual al directorio `bin` en `jakarta-tomcat-4.1.12`. En este directorio se encuentran los archivos `startup.bat`, `shutdown.bat`, `startup.sh` y `shutdown.sh`, para iniciar y detener el servidor Tomcat en Windows y UNIX/Linux/Mac OS X, respectivamente. Para iniciar el servidor, escriba:

```
startup
```

Esto inicia el servidor Tomcat, el cual se ejecuta en el puerto TCP 8080 para evitar conflictos con los servidores Web estándar que generalmente se ejecutan en el puerto TCP 80. Para verificar que Tomcat se esté ejecutando y pueda responder a las peticiones, abra su navegador Web y escriba el siguiente URL:

```
http://localhost:8080/
```

Esto debe mostrar la página inicial de documentación de Tomcat (figura 24.7). El host `localhost` indica al navegador Web que debe pedir la página inicial al servidor Tomcat del equipo local.

Si la página inicial de documentación de Tomcat no aparece en pantalla, pruebe con el siguiente URL:

```
http://127.0.0.1:8080/
```

El host `localhost` se traduce en la dirección IP `127.0.0.1`.

Tip para prevenir errores 24.2

Si el nombre de host `localhost` no funciona en su equipo, sustitúyalo por la dirección IP `127.0.0.1`.

Para detener el servidor Tomcat, ejecute el siguiente comando:

```
shutdown
```

desde una ventana de símbolo del sistema (o shell).

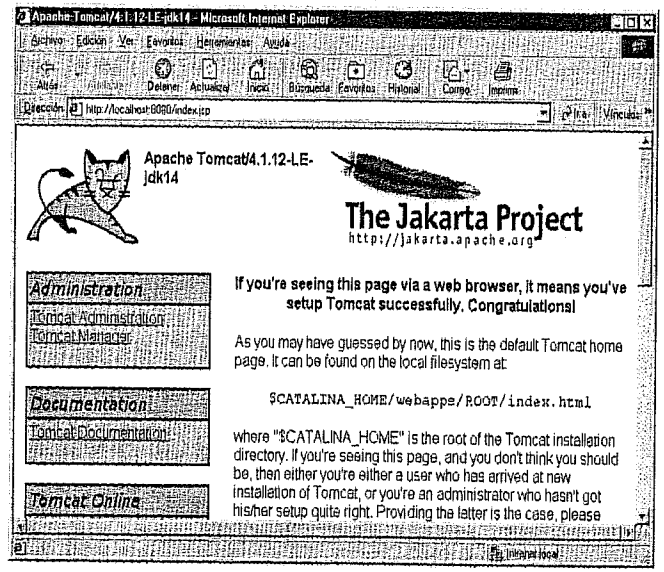


Figura 24.7 Página inicial de documentación de Tomcat. (Cortesía de la Fundación de software Apache.)

http://librosysolucionarios.net

24.3.2 Despliegue de una aplicación Web

Las JSPs, los servlets y sus archivos de soporte se despliegan como parte de *aplicaciones Web*. Generalmente, las aplicaciones Web se despliegan en el subdirectorio **webapps** de **jakarta-tomcat-4.1.12**. Una aplicación Web tiene una estructura de directorios reconocida, en la que residen todos los archivos que forman parte de la aplicación. Esta estructura de directorios puede ser creada por el administrador del servidor en el directorio **webapps**, o toda la estructura de directorios completa puede archivarse en un *archivo de fichero de aplicación Web*. Dicho archivo se conoce como *archivo WAR* y termina con la extensión de archivo **.war**. Si un archivo WAR se coloca en el directorio **webapps**, entonces cuando el servidor Tomcat empiece a ejecutarse, extraerá el contenido del archivo WAR en la estructura de subdirectorios apropiada de **webapps**. Por cuestiones de simpleza al enseñar sobre servlets y JavaServer Pages, creamos la estructura de directorios ya expandida para todos los ejemplos en este capítulo.

La estructura de directorios de aplicaciones Web contiene un directorio *raíz de contexto* (el directorio de nivel superior para toda una aplicación Web) y varios subdirectorios. Éstos se describen en la figura 24.8.



Error común de programación 24.1

El uso de "servlet" o "servlets" como raíz de contexto puede evitar que un servlet funcione correctamente en algunos servidores.

Para configurar el directorio raíz de contexto para una aplicación Web en Tomcat se requiere la creación de un subdirectorio en el directorio **webapps**. Cuando Tomcat empieza a ejecutarse, crea un directorio raíz de contexto para cada subdirectorio de **webapps**, utilizando el nombre de cada subdirectorio como un nombre raíz de contexto. Para probar los ejemplos en este capítulo, cree el directorio **cpej5** en el directorio **webapps** de Tomcat.

Después de configurar el directorio raíz de contexto, debemos configurar nuestra aplicación Web para manejar las peticiones. Esta configuración se lleva a cabo en un *descriptor de despliegue*, el cual se almacena en un archivo llamado **web.xml**. El descriptor de despliegue especifica varios parámetros de configuración como el nombre usado para invocar el servlet (es decir, su *alias*), una descripción del servlet, el nombre de clase completamente calificado del servlet y una *asociación de servlet* (es decir, la ruta o rutas que hacen que el contenedor de servlets invoque al servlet). Usted debe crear el archivo **web.xml** para este ejemplo. Muchas herramientas de despliegue de aplicaciones Web de Java crean el archivo **web.xml** por usted. El archivo **web.xml** para el primer ejemplo de este capítulo se muestra en la figura 24.9. Modificaremos este archivo a medida que agreguemos más servlets a la aplicación Web a lo largo de este capítulo.

Directorio	Descripción
raíz de contexto	Éste es el directorio raíz para la aplicación Web. Todas las JSPS, documentos de HTML, servlets y archivos de soporte como imágenes y archivos de clases residen en este directorio o sus subdirectorios. El nombre de este directorio es especificado por el creador de la aplicación Web. Para proporcionar la estructura en una aplicación Web, los subdirectorios pueden colocarse en el directorio raíz de contexto. Por ejemplo, si su aplicación utiliza muchas imágenes, podría colocar un subdirectorio llamado **imagenes** en este directorio. En los ejemplos de este capítulo utilizamos **cpej5** como el directorio raíz de contexto.
WEB-INF	Este directorio contiene el *descriptor de despliegue* de la aplicación Web (**web.xml**).
WEB-INF/classes	Este directorio contiene los archivos de clases del servlet y demás archivos de clases de soporte que se utilizan en una aplicación Web. Si las clases son parte de un paquete, la estructura de directorio completa del paquete empezaría aquí.
WEB-INF/lib	Este directorio contiene archivos de fichero de Java (JAR). Los archivos JAR pueden contener archivos de clases de servlets y demás archivos de clases de soporte que se utilizan en una aplicación Web.

Figura 24.8 Directorios estándar de aplicaciones Web.

http://librosysolucionarios.net


```
1  <!DOCTYPE web-app PUBLIC
2     "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
3     "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
4
5  <web-app>
6
7     <!-- Descripcion general de su aplicacion Web -->
8     <display-name>
9        Ejemplos del capitulo sobre JSP y Servlets
10       de Como programar en Java
11    </display-name>
12
13    <description>
14       Esta es la aplicacion Web en la que
15       demostramos nuestros ejemplos de JSP y Servlets
16    </description>
17
18    <!-- Definiciones de Servlets -->
19    <servlet>
20       <servlet-name>bienvenida1</servlet-name>
21
22       <description>
23          Un servlet simple que maneja una peticion get de HTTP
24       </description>
25
26       <servlet-class>
27          ServletDeBienvenida
28       </servlet-class>
29    </servlet>
30
31    <!-- Asociaciones de Servlets -->
32    <servlet-mapping>
33       <servlet-name>bienvenida1</servlet-name>
34       <url-pattern>/bienvenida1</url-pattern>
35    </servlet-mapping>
36
37 </web-app>
```

Figura 24.9 Descriptor de despliegue (**web.xml**) para la aplicación Web **cpej5**.

En las líneas 1 a 3 se especifica el tipo de documento para el descriptor de despliegue de la aplicación Web y la ubicación de la DTD para este archivo de XML. El elemento ***web-app*** (líneas 5 a 37) define la configuración de cada servlet en la aplicación Web y la asociación de servlet para cada servlet. El elemento ***display-name*** (líneas 8 a 11) especifica un nombre que puede mostrarse al administrador del servidor en el que está instalada la aplicación Web. El elemento ***description*** (líneas 13 a 16) especifica una descripción de la aplicación Web que podría mostrarse al administrador del servidor.

El elemento **servlet** (líneas 19 a 29) describe un servlet. El elemento ***servlet-name*** (línea 20) es el nombre que elegimos para el servlet (**bienvenida1**). El elemento ***description*** (líneas 22 a 24) especifica una descripción para este servlet específico. De nuevo, éste puede mostrarse al administrador del servidor Web. El elemento **servlet-class** (líneas 26 a 28) especifica el nombre de clase completamente calificado del servlet compilado. Por lo tanto, el servlet **bienvenida1** está definido por la clase **ServletDeBienvenida**.

El elemento ***servlet-mapping*** (líneas 32 a 35) especifica los elementos ***servlet-name*** y ***url-pattern***. El *patrón de URL* ayuda al servidor a determinar cuáles peticiones se envían al servlet (**bienvenida1**). Nuestra aplicación Web se instalará como parte del directorio raíz de contexto **cpej5** descrito en la

sección 24.3.2. Por lo tanto, el URL relativo que proporcionamos al navegador para invocar al servlet en este ejemplo es:

```
/cpej5/bienvenida1
```

en donde `/cpej5` especifica el directorio raíz de contexto que ayuda al servidor a determinar cuál aplicación Web va a manejar la petición, y `/bienvenida1` especifica el patrón de URL que se asocia al servlet `bienvenida1` para manejar la petición. Observe que el servidor en el que reside el servlet no se especifica aquí, aunque es posible hacerlo de la siguiente manera:

```
http://localhost:8080/cpej5/bienvenida1
```

Si el servidor y número de puerto explícitos no se especifican como parte del URL, el navegador supone que el manejador de formularios (es decir, el servlet especificado en la propiedad `action` del elemento `form`) reside en el mismo servidor y número de puerto del cual el navegador descargó la página Web que contiene el elemento `form`.

Hay varios formatos de patrón de URL que pueden usarse. El patrón de URL `/bienvenida1` requiere una concordancia exacta del patrón. También puede especificar *asociaciones de rutas*, asociaciones de extensiones y un *servlet predeterminado* para una aplicación Web. Una asociación de ruta empieza con una `/` y termina con una `/*`. Por ejemplo, el patrón de URL:

```
/cpej5/ejemplo/*
```

indica que cualquier ruta de URL que empiece con `/cpej5/ejemplo` se enviará al servlet que tenga el patrón de URL anterior. Una asociación de extensión empieza con `*.` y termina con una extensión de nombre de archivo. Por ejemplo, el patrón de URL:

```
*.jsp
```

indica que cualquier petición de un archivo con extensión `.jsp` se enviará al servlet que maneje peticiones de JSP. De hecho, los servlets con contenedores de JSP tienen una asociación implícita de la extensión `.jsp` a un servlet que maneja peticiones de JSP. El patrón de URL `/` representa el servlet predeterminado para la aplicación Web. Esto es similar al documento predeterminado de un servidor Web. Por ejemplo, si usted escribe el URL `www.deitel.com` en su navegador Web, el documento que recibe de nuestro servidor Web es el documento predeterminado `index.html`. Si el patrón de URL concuerda con el servlet predeterminado para una aplicación Web, ese servlet se invoca para devolver una respuesta predeterminada al cliente. Esto puede ser útil para personalizar contenido Web para usuarios específicos.

Finalmente, estamos listos para colocar nuestros archivos en los directorios apropiados para completar el despliegue de nuestro primer servlet y probarlo. Hay tres archivos que debemos colocar en los directorios apropiados: `ServletDeBienvenida.html`, `ServletDeBienvenida.class` y `web.xml`. En el subdirectorio `webapps` de su directorio `jakarta-tomcat-4.1.12`, cree el subdirectorio `cpej5` (el directorio raíz de contexto para nuestra aplicación). En este directorio, cree los subdirectorios llamados `servlets` y `WEB-INF`. Colocaremos nuestros archivos de HTML para los servlets de este capítulo en el directorio `servlets`. Copie el archivo `ServletDeBienvenida.html` en el directorio `servlets`. En el directorio `WEB-INF`, cree el subdirectorio `classes` y después copie el archivo `web.xml` en el directorio `WEBINF`, y copie el archivo `ServletDeBienvenida.class` en el directorio `classes`. Entonces, la estructura de directorios y archivos bajo el directorio `webapps` debe ser similar a la que se muestra en la figura 24.10 (los nombres de archivo están en cursiva).

Tip para prevenir errores 24.3



Reinicie el servidor Tomcat después de modificar el archivo descriptor de despliegue `web.xml`. Si no lo hace, Tomcat no reconocerá su nueva aplicación Web.

Una vez que los archivos se coloquen en los directorios apropiados, inicie el servidor Tomcat, abra su navegador y escriba el siguiente URL:

```
http://localhost:8080/cpej5/servlets/ServletDeBienvenida.html
```

para cargar el archivo `ServletDeBienvenida.html` en el navegador Web. Después, haga clic en el botón **Obtener documento de HTML** para invocar el servlet. Deberá ver los resultados que aparecen en la figura 24.6.

Estructura de directorios y archivos de la aplicación Web `ServletDeBienvenida`

```
cpej5
   servlets
      ServletDeBienvenida.html
   WEB-INF
      web.xml
      classes
         ServletDeBienvenida.class
```

Figura 24.10 Estructura de directorios y archivos de la aplicación Web para `ServletDeBienvenida`.

Puede probar este servlet desde varios navegadores Web distintos para demostrar que los resultados son los mismos.

Error común de programación 24.2



Si no se colocan los archivos del servlet o demás archivos de clases en la estructura de directorios apropiada, el servlet no podrá localizar esas clases correctamente. Esto produce una respuesta de error en el navegador Web. Esta respuesta de error es normalmente "Not Found (404)" (No encontrado) en Netscape Navigator y "No se encontró la página" junto con una explicación en Microsoft Internet Explorer.

En realidad, el archivo de HTML de la figura 24.6 no es necesario para invocar este servlet. Una petición `get` puede enviarse a un servidor con sólo escribir el URL en un navegador; de la misma manera que como cuando se pide una página Web en un navegador. En este ejemplo, usted puede escribir:

```
http://localhost:8080/cpej5/bienvenida1
```

en el campo de **dirección** de su navegador para invocar el servlet directamente.

Tip para prevenir errores 24.4



Puede probar un servlet que maneje peticiones `get` de HTTP escribiendo el URL que invoca al servlet directamente en el **campo** *de* **dirección** *de su navegador.*

24.4 Manejo de peticiones `get` de HTTP que contienen datos

Al pedir un documento o recurso de un servidor Web, es posible suministrar datos como parte de la petición. El servlet `ServletDeBienvenida2` de la figura 24.11 responde a una petición `get` de HTTP que contiene un nombre suministrado por el usuario. Este servlet utiliza el nombre como parte de la respuesta al cliente.

```
1  // Fig. 24.11: ServletDeBienvenida2.java
2  // Procesamiento de peticiones get de HTTP que contienen datos.
3
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.io.*;
7
8  public class ServletDeBienvenida2 extends HttpServlet {
9
10    // procesar petición "get" del cliente
11    protected void doGet( HttpServletRequest peticion,
12       HttpServletResponse respuesta )
13          throws ServletException, IOException
```

Figura 24.11 `ServletDeBienvenida2` responde a una petición `get` con datos. (Parte 1 de 2.)

http://librosysolucionarios.net

```
14    {
15       String nombrePila = peticion.getParameter( "nombrepila" );
16
17       respuesta.setContentType( "text/html" );
18       PrintWriter salida = respuesta.getWriter();
19
20       // enviar documento de XHTML al cliente
21
22       // empezar documento de XHTML
23       salida.println( "<?xml version = \"1.0\"?>" );
24
25       salida.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
26          "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
27          "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
28
29       salida.println( "<html xmlns = \"http://www.w3.org/1999/xhtml\">" );
30
31       // sección de encabezado del documento
32       salida.println( "<head>" );
33       salida.println(
34          "<title>Procesamiento de peticiones get con datos</title>" );
35       salida.println( "</head>" );
36
37       // sección del cuerpo del documento
38       salida.println( "<body>" );
39       salida.println( "<h1>Hola " + nombrePila + "<br />" );
40       salida.println( "¡Bienvenido a los servlets!</h1>" );
41       salida.println( "</body>" );
42
43       // fin del documento de XHTML
44       salida.println( "</html>" );
45       salida.close();  // cerrar flujo para completar la página
46    }
47 }
```

Figura 24.11 **ServletDeBienvenida2** responde a una petición **get** con datos. (Parte 2 de 2.)

En una petición **get**, los parámetros se pasan como pares nombre/valor. En la línea 15 se demuestra cómo obtener información que se pasa al servlet como parte de la petición del cliente. El método ***getParameter*** del objeto **petición** recibe el nombre del parámetro como un argumento y devuelve el valor **String** correspondiente, o **null** si el parámetro no forma parte de la petición. En la línea 39 se utiliza el resultado de la línea 15 como parte de la respuesta al cliente.

El documento **ServletDeBienvenida2.html** (figura 24.12) proporciona un elemento **form** en el que el usuario puede escribir un nombre en el elemento **nombrepila** de entrada (**input**) de texto (línea 17) y hacer clic en el botón **Enviar** para invocar a **ServletDeBienvenida2**. Cuando el usuario oprime el botón **Enviar**, los valores de los elementos **input** se colocan en pares nombre/valor como parte de la petición al servidor. En la segunda captura de pantalla de la figura 24.12, observe que el navegador anexó lo siguiente:

```
?nombrepila=Paul
```

al final del URL de la propiedad **actino**. El **?** separa la *cadena de consulta* (es decir, los datos que se pasan como parte de la petición **get**) del resto del URL en una petición **get**. Los pares nombre/valor se pasan con el nombre y el valor separados por un **=**. Si hay más de un par nombre/valor, cada par se separa por un **&**.

Una vez más, utilizamos nuestro directorio raíz de contexto **cpej5** para demostrar el servlet de la figura 24.11. Coloque a **ServletDeBienvenida2.html** en el directorio **servlets** creado en la sección 24.3.2. Coloque a **ServletDeBienvenida2.class** en el subdirectorio **classes** de **WEB-INF** en el directorio

http://librosysolucionarios.net


```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 24.12: ServletDeBienvenida2.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Procesamiento de peticiones get con datos</title>
10 </head>
11
12 <body>
13    <form action = "/cpej5/bienvenida2" method = "get">
14
15       <p><label>
16          Escriba su nombre de pila y oprima el botón Enviar
17          <br /><input type = "text" name = "nombrepila" />
18          <input type = "submit" value = "Enviar" />
19       </p></label>
20
21    </form>
22 </body>
23 </html>
```

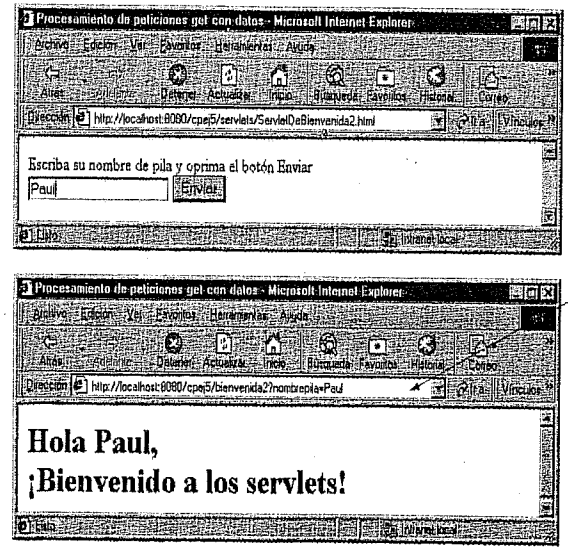


Figura 24.12 Documento de HTML en el que la propiedad **action** del elemento **form** invoca a **ServletDeBienvenida2** a través del alias **bienvenida2** especificado en **web.xml**.

raíz de contexto **cpej5**. Recuerde que las clases en un paquete deben colocarse en la estructura de directorios apropiada para ese paquete. Después edite el descriptor de despliegue **web.xml** en el directorio **WEB-INF** para incluir la información especificada en la figura 24.13. Esta tabla contiene la información para los elementos **servlet** y **servlet-mapping** que usted agregará al descriptor de despliegue **web.xml**. No debe escri-

Elemento del descriptor	Valor
elemento `servlet`	
`servlet-name`	`bienvenida2`
`description`	`Manejo de peticiones get de HTTP con datos.`
`servlet-class`	`ServletDeBienvenida2`
elemento `servlet-mapping`	
`servlet-name`	`bienvenida2`
`url-pattern`	`/bienvenida2`

Figura 24.13 Información del descriptor de despliegue para el servlet `ServletDeBienvenida2`.

bir el texto en cursiva en el descriptor de despliegue. Reinicie Tomcat y escriba el siguiente URL en su navegador Web:

```
http://localhost:8080/cpej5/servlets/ServletDeBienvenida2.html
```

Escriba su nombre en el campo de texto de la página Web y después haga clic en **Enviar** para invocar el servlet.

Una vez más, observe que la petición `get` hubiera podido escribirse directamente en el **campo de dirección** del navegador, de la siguiente manera:

```
http://localhost:8080/cpej5/bienvenida2?nombrepila=Paul
```

Pruébelo con su propio nombre.

24.5 Manejo de las peticiones `post` de HTTP

Una petición `post` de HTTP se utiliza comúnmente para enviar datos de un formulario de HTML a un manejador de formularios del lado servidor que procesa los datos. Por ejemplo, cuando usted responde a una encuesta basada en Web, una petición `post` normalmente suministra la información que usted escribe en el formulario a un servidor Web.

Los navegadores a menudo colocan en *caché* (almacenan en disco) las páginas Web, para que puedan volver a cargarlas rápidamente. Si no hay cambios entre la última versión almacenada en la caché y la versión actual en Web, esto ayuda a agilizar su experiencia de navegación. El navegador primero pregunta al servidor si el documento ha cambiado o expirado desde la fecha en que el archivo se colocó en la caché. De no ser así, el navegador carga el documento de la caché. Por lo tanto, el navegador minimiza la cantidad de datos que deben descargarse para que usted pueda ver una página Web. Generalmente los navegadores no colocan en caché la respuesta del servidor a una petición `post`, ya que la siguiente petición `post` podría no devolver el mismo resultado. Por ejemplo, en una encuesta muchos usuarios podrían visitar la misma página Web y responder a una pregunta. Los resultados de la encuesta podrían entonces mostrarse al usuario. Cada nueva respuesta cambia los resultados totales de la encuesta.

Cuando usted utiliza un motor de búsqueda basado en Web, el navegador normalmente suministra la información que usted le especifica en un formulario de HTML al motor de búsqueda, mediante una petición `get`. El motor de búsqueda realiza la búsqueda y después le devuelve a usted los resultados como una página Web. El navegador coloca dichas páginas frecuentemente en la caché, en caso de que usted realice la misma búsqueda de nuevo. Al igual que las peticiones `post`, las peticiones `get` pueden suministrar parámetros como parte de la petición al servidor Web.

El servlet `ServletDeBienvenida3` de la figura 24.14 es idéntico al de la figura 24.11, con la excepción de que define un método `doPost` (línea 11) para responder a las peticiones `post`, en vez de definir un método `doGet`. La funcionalidad predeterminada de `doPost` es indicar un error "Method not allowed" (Método no permitido). Redefinimos este método para proporcionar un procesamiento de petición `post` personalizado. El método `doPost` recibe los mismos dos argumentos que `doGet`: un objeto que implementa a la

interfaz `HttpServletRequest` para representar la petición del cliente y un objeto que implementa a la interfaz `HttpServletResponse` para representar la respuesta del servlet. Al igual que `doGet`, el método `doPost` lanza una excepción `ServletException` si no puede manejar la petición de un cliente y lanza una excepción `IOException` si ocurre un problema durante el procesamiento de los flujos.

`ServletDeBienvenida3.html` (figura 24.15) proporciona un elemento `form` (líneas 13 a 21) en el que el usuario puede escribir un nombre en el elemento `nombrepila` de entrada (`input`) de texto (línea 17) y después hacer clic en el botón **Enviar** para invocar a `ServletDeBienvenida3`. Cuando el usuario oprime el botón **Enviar**, los valores de los elementos `input` se envían al servidor como parte de la petición. Sin embargo, observe que los valores no se anexan al URL de la petición. Observe que la propiedad `method` del formulario en este ejemplo es `post`. Además, observe que una petición `post` no puede escribirse en el **campo de dirección** y que los usuarios no pueden marcar las peticiones `post` en sus navegadores.

```
1  // Fig. 24.14: ServletDeBienvenida3.java
2  // Procesamiento de peticiones post con datos.
3
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.io.*;
7
8  public class ServletDeBienvenida3 extends HttpServlet {
9
10    // procesar petición "post" del cliente
11    protected void doPost( HttpServletRequest peticion,
12       HttpServletResponse respuesta )
13          throws ServletException, IOException
14    {
15       String nombrePila = peticion.getParameter( "nombrepila" );
16
17       respuesta.setContentType( "text/html" );
18       PrintWriter salida = respuesta.getWriter();
19
20       // enviar página de XHTML al cliente
21
22       // empezar documento de XHTML
23       salida.println( "<?xml version = \"1.0\"?>" );
24
25       salida.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
26          "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
27          "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
28
29       salida.println( "<html xmlns = \"http://www.w3.org/1999/xhtml\">" );
30
31       // sección de encabezado del documento
32       salida.println( "<head>" );
33       salida.println(
34          "<title>Procesamiento de peticiones post con datos</title>" );
35       salida.println( "</head>" );
36
37       // sección del cuerpo del documento
38       salida.println( "<body>" );
39       salida.println( "<h1>Hola " + nombrePila + "<br />" );
40       salida.println( "¡Bienvenido a los servlets!</h1>" );
41       salida.println( "</body>" );
42
```

Figura 24.14 `ServletDeBienvenida3` responde a una petición `post` con datos. (Parte 1 de 2.)

http://librosysolucionarios.net

```
43         // fin del documento de XHTML
44         salida.println( "</html>" );
45         salida.close();  // cerrar flujo para completar la página
46      }
47   }
```

Figura 24.14 `ServletDeBienvenida3` responde a una petición **post** con datos. (Parte 2 de 2.)

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 24.15: ServletDeBienvenida3.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Manejo de una petición post de HTTP con datos</title>
10 </head>
11
12 <body>
13    <form action = "/cpej5/bienvenida3" method = "post">
14
15       <p><label>
16          Escriba su nombre de pila y oprima el botón Enviar
17          <br /><input type = "text" name = "nombrepila" />
18          <input type = "submit" value = "Enviar" />
19       </label></p>
20
21    </form>
22 </body>
23 </html>
```

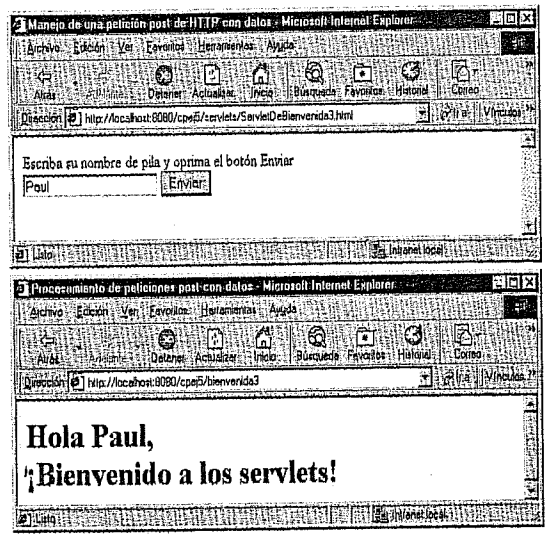


Figura 24.15 Documento de HTML en el que la propiedad **action** del formulario invoca a `ServletDeBienvenida3` a través del alias **bienvenida3** especificado en `web.xml`.

http://librosysolucionarios.net

Elemento del descriptor	Valor
elemento `servlet`	
`servlet-name`	`bienvenida3`
`description`	`Manejo de peticiones post de HTTP con datos.`
`servlet-class`	`ServletBienvenida3`
elemento `servlet-mapping`	
`servlet-name`	`bienvenida3`
`url-pattern`	`/bienvenida3`

Figura 24.16 Información del descriptor de despliegue para el servlet `ServletDeBienvenida3`.

Utilizamos nuestro directorio raíz de contexto `cpej5` para demostrar el servlet de la figura 24.14. Coloque a `ServletDeBienvenida3.html` en el directorio `servlets` creado en la sección 24.3.2. Coloque a `ServletDeBienvenida3.class` en el subdirectorio `classes` de `WEB-INF` en el directorio raíz de contexto `cpej5`. Después, utilizando la información especificada en la figura 24.16, edite el descriptor de despliegue `web.xml` en el directorio `WEB-INF`. Reinicie Tomcat y escriba el siguiente URL en su navegador Web:

```
http://localhost:8080/cpej5/servlets/ServletDeBienvenida3.html
```

Escriba su nombre en el campo de texto de la página Web y después haga clic en **Enviar** para invocar el servlet.

24.6 Redirección de peticiones a otros recursos

Algunas veces es útil redireccionar una petición hacia un recurso distinto. Por ejemplo, un servlet podría determinar el tipo del navegador del cliente y redireccionar la petición hacia una página Web que haya sido diseñada específicamente para ese navegador. El `ServletRedireccion` de la figura 24.17 recibe un parámetro de página como parte de una petición `get` y después lo utiliza para redireccionar la petición hacia un recurso distinto.

```
1  // Fig. 24.17: ServletRedireccion.java
2  // Redireccionar un usuario hacia una página Web distinta.
3
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.io.*;
7
8  public class ServletRedireccion extends HttpServlet {
9
10    // procesar petición "get" del cliente
11    protected void doGet( HttpServletRequest peticion,
12       HttpServletResponse respuesta )
13          throws ServletException, IOException
14    {
15       String ubicacion = peticion.getParameter( "pagina" );
16
17       if ( ubicacion != null )
18
19          if ( ubicacion.equals( "deitel" ) )
20             respuesta.sendRedirect( "http://www.deitel.com" );
```

Figura 24.17 Redireccionamiento de peticiones hacia otros recursos. (Parte 1 de 2.)

```
21          else
22             if ( ubicacion.equals( "bienvenida1" ) )
23                respuesta.sendRedirect( "bienvenida1" );
24
25       // código que se ejecuta solamente si este servlet
26       // no redirecciona al usuario hacia otra página
27
28       respuesta.setContentType( "text/html" );
29       PrintWriter salida = respuesta.getWriter();
30
31       // empezar documento de XHTML
32       salida.println( "<?xml version = \"1.0\"?>" );
33
34       salida.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
35          "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
36          "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
37
38       salida.println(
39          "<html xmlns = \"http://www.w3.org/1999/xhtml\">" );
40
41       // sección de encabezado del documento
42       salida.println( "<head>" );
43       salida.println( "<title>Página inválida</title>" );
44       salida.println( "</head>" );
45
46       // sección del cuerpo del documento
47       salida.println( "<body>" );
48       salida.println( "<h1>Solicitó una página inválida</h1>" );
49       salida.println( "<p><a href = " +
50          "\"servlets/ServletRedireccion.html\">" );
51       salida.println( "Haga clic aquí para seleccionar de nuevo</a></p>" );
52       salida.println( "</body>" );
53
54       // fin del documento de XHTML
55       salida.println( "</html>" );
56       salida.close();  // cerrr flujo para completar la página
57    }
58 }
```

Figura 24.17 Redireccionamiento de peticiones hacia otros recursos. (Parte 2 de 2.)

En la línea 15 se obtiene el parámetro `pagina` de la petición. Si el valor devuelto no es `null`, la estructura `if...else` de las líneas 19 a 23 determina si el valor es `"deitel"` o `"bienvenida1"`. Si el valor es `"deitel"`, el método ***sendRedirect*** del objeto `respuesta` (línea 20) redirecciona la petición hacia `www.deitel.com`. Si el valor es `"bienvenida1"`, en la línea 23 se redirecciona la petición hacia el servlet de la figura 24.5. Observe que en la línea 23 no se especifica explícitamente el directorio raíz de contexto `cpej5` para nuestra aplicación Web. Cuando un servlet utiliza una ruta relativa para hacer referencia a otro recurso estático o dinámico, el servlet supone que se utilizan los mismos URLs base y directorio raíz de contexto que se usaron para invocar al servlet; a menos que se especifique un URL completo para el recurso. Por lo tanto, en la línea 23 se está pidiendo el recurso que se encuentra en:

```
http://localhost:8080/cpej5/bienvenida1
```

De manera similar, en la línea 50 se está pidiendo el recurso ubicado en:

```
http://localhost:8080/cpej5/servlets/ServletRedireccion.html
```

http://librosysolucionarios.net

Observación de ingeniería de software 24.5



El uso de rutas relativas para hacer referencia a los recursos en el mismo directorio raíz de contexto hace que su aplicación Web sea más flexible. Por ejemplo, puede cambiar el directorio raíz de contexto sin tener que hacer cambios en los recursos estáticos y dinámicos de la aplicación.

Una vez que se ejecuta el método `sendRedirect`, `ServletRedireccion` termina el procesamiento de la petición original. Si el método `sendRedirect` no es llamado, el resto del método `doGet` muestra una página Web en la que se indica que se hizo una petición inválida. La página permite al usuario intentar de nuevo, regresando al documento de XHTML de la figura 24.18. Observe que uno de los redireccionamientos es hacia una página Web de XHTML estática, y el otro es hacia un servlet.

`ServletRedireccion.html` (figura 24.18) proporciona dos hipervínculos (líneas 15 a 16 y 17 a 18) que permiten al usuario invocar el servlet `ServletRedireccion`. Observe que cada hipervínculo especifica un parámetro `pagina` como parte del URL. Para mostrar cómo se pasa una página inválida, puede escribir el URL en su navegador sin ningún valor para el parámetro `pagina`.

Utilizamos nuestro directorio raíz de contexto `cpej5` para mostrar el servlet de la figura 24.17. Coloque a `ServletRedireccion.html` en el directorio `servlets` creado en la sección 24.3.2. Coloque a `ServletRedireccion.class` en el subdirectorio `classes` de `WEB-INF` en el directorio raíz de con-

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 24.18: ServletRedireccion.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Redireccionar una petición hacia otro sitio</title>
10 </head>
11
12 <body>
13    <p>Haga clic en un vínculo para redireccionarlo a la página apropiada</p>
14    <p>
15    <a href = "/cpej5/redireccion?pagina=deitel">
16       www.deitel.com</a><br />
17    <a href = "/cpej5/redireccion?pagina=bienvenida1">
18       Servlet de bienvenida</a>
19    </p>
20 </body>
21 </html>
```

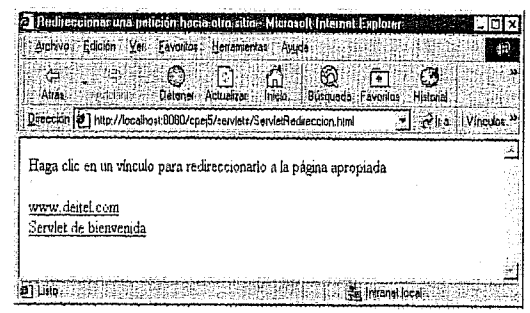


Figura 24.18 Documento `ServletRedireccion.html` para demostrar cómo redireccionar peticiones hacia otros recursos. (Parte 1 de 2.)

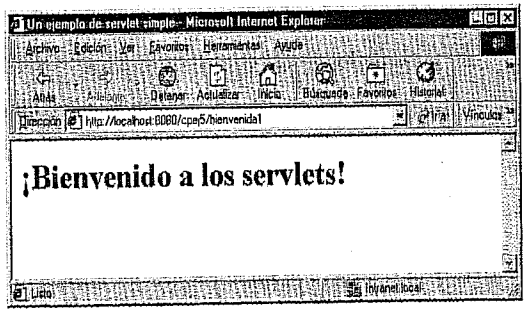


Figura 24.18 Documento `ServletRedireccion.html` para demostrar cómo redireccionar peticiones hacia otros recursos. (Parte 2 de 2.)

Elemento del descriptor	Valor
elemento `servlet`	
`servlet-name`	`redireccion`
`description`	`Redireccionamiento hacia paginas Web estaticas y otros servlets.`
`servlet-class`	`ServletRedireccion`
elemento `servlet-mapping`	
`servlet-name`	`redireccion`
`url-pattern`	`/redireccion`

Figura 24.19 Información del descriptor de despliegue para el servlet `ServletRedireccion`.

texto `cpej5`. Después edite el descriptor de despliegue `web.xml` en el directorio `WEB-INF` para incluir la información especificada en la figura 24.19. Reinicie Tomcat y escriba el siguiente URL en su navegador Web:

```
http://localhost:8080/cpej5/servlets/ServletRedireccion.html
```

Haga clic en uno de los hipervínculos de la página Web para invocar el servlet.

Al redireccionar peticiones, los parámetros de la petición original se pasan como parámetros a la nueva petición. También pueden pasarse parámetros de petición adicionales. Por ejemplo, el URL que se pasa a `sendRedirect` podría contener pares nombre/valor. Los nuevos parámetros se agregan a los ya existentes. Un nuevo parámetro con el mismo nombre que uno existente tiene precedencia sobre el valor original. Sin embargo, todos los valores se pasan de todas formas. En este caso, puede obtenerse el conjunto completo de valores para un nombre de parámetro dado mediante una llamada al método ***`getParameterValues`*** de la interfaz `HttpServletRequest`. Este método recibe el nombre del parámetro como un argumento y devuelve un arreglo de cadenas que contienen los valores del parámetro, en orden del más al menos reciente .

24.7 Aplicaciones multinivel: Uso de JDBC desde un servlet

Los servlets pueden comunicarse con las bases de datos a través de JDBC. Como lo dijimos en el capítulo 23, JDBC proporciona una manera uniforme para que los programas de Java se conecten a una variedad de bases de datos en forma general, sin tener que tratar con los detalles específicos de esos sistemas de bases de datos.

Muchas de las aplicaciones de la actualidad son *aplicaciones distribuidas de tres niveles*, las cuales consisten en una *interfaz de usuario*, la *lógica de negocios* y una *base de datos*. La interfaz de usuario en dicha aplicación se crea a menudo utilizando HTML, XHTML (como se muestra en este capítulo) o HTML dinámico. HTML y XHTML son los mecanismos preferidos para representar la interfaz de usuario en sistemas en los que la portabilidad es una cuestión primordial. Como todos los navegadores soportan HTML, al diseñar la interfaz de usuario para que pueda utilizarse a través de un navegador Web se garantiza la portabilidad entre todas las plataformas que tengan navegadores. Utilizando los servicios de red proporcionados por el navegador, la interfaz de usuario se puede comunicar con la lógica de negocios del nivel intermedio. Este nivel puede entonces acceder a la base de datos para manipular la información. Los tres niveles pueden residir en equipos separados que estén conectados a una red.

En arquitecturas de multinivel, a menudo se utilizan servidores Web en el nivel intermedio. Los componentes del lado servidor (como los servlets) se ejecutan en un *servidor de aplicaciones* junto con el servidor Web. Estos componentes proporcionan la lógica de negocios que manipula la información de las bases de datos y que se comunica con los navegadores Web clientes. Mediante JDBC, los servlets pueden interactuar con los sistemas de bases de datos populares. Los desarrolladores utilizan consultas basadas en SQL y controladores JDBC para manejar los detalles específicos de la interacción con cada sistema de bases de datos.

El `ServletEncuesta` de la figura 24.20 y el documento `Encuesta.html` de la figura 24.21 implementan partes de una aplicación distribuida de tres niveles. El nivel intermedio es `ServletEncuesta`, el cual maneja las peticiones del navegador cliente y proporciona acceso al tercer nivel: una base de datos de Cloudscape a la que se tiene acceso mediante JDBC. El servlet en este ejemplo permite a los usuarios votar por sus animales favoritos. Cuando el servlet recibe una petición `post` del navegador Web, utiliza JDBC para actualizar el número total de votos para ese animal en la base de datos y devuelve al cliente un documento de XHTML generado en forma dinámica, el cual contiene los resultados de la encuesta.

```
1  // Fig. 24.20: ServletEncuesta.java
2  // Una encuesta basada en Web que utiliza JDBC desde un servlet.
3  package com.deitel.cpej5.servlets;
4
5  import java.io.*;
6  import java.text.*;
7  import java.sql.*;
8  import javax.servlet.*;
9  import javax.servlet.http.*;
10
11 public class ServletEncuesta extends HttpServlet {
12    private Connection conexion;
13    private Statement instruccion;
14
15    // establecer conexión a la base de datos y crear instrucción de SQL
16    public void init( ServletConfig config ) throws ServletException
17    {
18       // intentar conexión a la base de datos y crear objeto Statement
19       try {
20          System.setProperty( "db2j.system.home",
21             config.getInitParameter( "ubicacionBaseDeDatos" ) );
22
23          Class.forName( config.getInitParameter( "controladorBaseDeDatos" ) );
24          conexion = DriverManager.getConnection(
25             config.getInitParameter( "nombreBaseDeDatos" ) );
26
27          // crear objeto Statement para consultar la base de datos
28          instruccion = conexion.createStatement();
29       }
30
```

Figura 24.20 Encuesta basada en Web con arquitectura de multinivel, la cual utiliza XHTML, servlets y JDBC. (Parte 1 de 4.)

http://librosysolucionarios.net

```
31          // para cualquier excepción, lanzar UnavailableException para
32          // indicar que el servlet no se encuentra disponible en ese momento
33          catch ( Exception excepcion ) {
34             excepcion.printStackTrace();
35             throw new UnavailableException( excepcion.getMessage() );
36          }
37
38       }  // fin del método init
39
40       // procesar respuesta de la encuesta
41       protected void doPost( HttpServletRequest peticion,
42          HttpServletResponse respuesta )
43             throws ServletException, IOException
44       {
45         // establecer respuesta para el cliente
46          respuesta.setContentType( "text/html" );
47          PrintWriter salida = respuesta.getWriter();
48          DecimalFormat dosDigitos = new DecimalFormat( "0.00" );
49
50          // empezar documento de XHTML
51          salida.println( "<?xml version = \"1.0\"?>" );
52
53          salida.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
54             "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
55             "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
56
57          salida.println(
58             "<html xmlns = \"http://www.w3.org/1999/xhtml\">" );
59
60          // sección de encabezado del documento
61          salida.println( "<head>" );
62
63          // leer respuesta actual de la encuesta
64          int valor =
65             Integer.parseInt( peticion.getParameter( "animal" ) );
66          String consulta;
67
68          // tratar de procesar un voto y mostrar los resultados actuales
69          try {
70
71             // actualizar el total para la respuesta actual de la encuesta
72             consulta = "UPDATE resultadosencuesta SET votos = votos + 1 " +
73                   "WHERE id = " + valor;
74             instruccion.executeUpdate( consulta );
75
76             // obtener el total de todas las respuestas de la encuesta
77             consulta = "SELECT sum( votos ) FROM resultadosencuesta";
78             ResultSet totalRS = instruccion.executeQuery( consulta );
79             totalRS.next();
80             int total = totalRS.getInt( 1 );
81
82             // obtener resultados
83             consulta = "SELECT opcionencuesta, votos, id FROM resultadosencuesta " +
84                "ORDER BY id";
```

Figura 24.20 Encuesta basada en Web con arquitectura de multinivel, la cual utiliza XHTML, servlets y JDBC. (Parte 2 de 4.)


```
85          ResultSet resultadosRS = instruccion.executeQuery( consulta );
86          salida.println( "<title>¡Gracias!</title>" );
87          salida.println( "</head>" );
88
89          salida.println( "<body>" );
90          salida.println( "<p>Gracias por su participación." );
91          salida.println( "<br />Resultados:</p><pre>" );
92
93          // procesar los resultados
94          int votos;
95
96          while ( resultadosRS.next() ) {
97             salida.print( resultadosRS.getString( 1 ) );
98             salida.print( ": " );
99             votos = resultadosRS.getInt( 2 );
100            salida.print( dosDigitos.format(
101               ( double ) votos / total * 100 ) );
102            salida.print( "%  respuestas: " );
103            salida.println( votos );
104         }
105
106         resultadosRS.close();
107
108         salida.print( "Total de respuestas: " );
109         salida.print( total );
110
111         // fin del documento de XHTML
112         salida.println( "</pre></body></html>" );
113         salida.close();
114
115      } // fin de bloque try
116
117      // si ocurre una excepción en la base de datos, devolver página de error
118      catch ( SQLException excepcionSQL ) {
119         excepcionSQL.printStackTrace();
120         salida.println( "<title>Error</title>" );
121         salida.println( "</head>" );
122         salida.println( "<body><p>Ocurrió un error en la base de datos. " );
123         salida.println( "Intente de nuevo más tarde.</p></body></html>" );
124         salida.close();
125      }
126
127   }  // fin del método doPost
128
129   // cerrar instrucciones de SQL y base de datos cuando termine el servlet
130   public void destroy()
131   {
132      // tratar de cerrar instrucciones y conexion a la base de datos
133      try {
134         instruccion.close();
135         conexion.close();
136      }
137
138      // manejar excepciones de la base de datos devolviendo el error al cliente
```

Figura 24.20 Encuesta basada en Web con arquitectura de multinivel, la cual utiliza XHTML, servlets y JDBC. (Parte 3 de 4.)

http://librosysolucionarios.net

```
139         catch( SQLException excepcionSQL ) {
140            excepcionSQL.printStackTrace();
141         }
142      }
143
144   } // fin de la clase ServletEncuesta
```

Figura 24.20 Encuesta basada en Web con arquitectura de multinivel, la cual utiliza XHTML, servlets y JDBC. (Parte 4 de 4.)

En las líneas 12 y 13 declaran una referencia **Connection** para manejar la conexión a la base de datos, y una referencia **Statement** para actualizar la cuenta de votos para un animal, totalizar todos los votos y obtener los resultados completos de la encuesta.

Los servlets se inicializan mediante el método ***init***, que sobrescribimos en **ServletEncuesta** (líneas 16 a 38). El método **init** es llamado solamente una vez durante la vida de un servlet, antes de aceptar cualquier petición de los clientes. El método **init** toma un argumento ***ServletConfig*** y lanza una excepción **ServletException**. El argumento proporciona información al servlet acerca de sus *parámetros de inicialización* (es decir, los parámetros que no están asociados con una petición, sino que se pasan al servlet para inicializar su estado). Estos parámetros se especifican en el archivo descriptor de despliegue **web.xml** como parte de un elemento **servlet**. Cada parámetro aparece en un elemento ***init-param*** de la siguiente forma:

```
<init-param>
   <param-name>nombre del parámetro</param-name>
   <param-value>nombre del parámetro</param-value>
</init-param>
```

Los servlets pueden obtener los valores de los parámetros de inicialización invocando al método ***getInitParameter*** de **ServletConfig**, el cual recibe una cadena que representa el nombre del parámetro.

En este ejemplo, el método **init** del servlet (líneas 16 a 38) realiza la conexión a la base de datos de Cloudscape. En las líneas 20 y 21 se invoca el método **setProperty** de **System** para asignar a la propiedad **"db2j.system.home"** la ubicación de la base de datos en el sistema de archivos. En la línea 21 se invoca el método **getInitParameter** para obtener el valor del parámetro de inicialización llamado **"ubicacionBaseDeDatos"**. (Para este ejemplo, creamos la base de datos en la carpeta **C:\CloudScape_5.1**). En la línea 23 se carga el controlador (**com.ibm.db2j.jdbc.DB2jDriver**, que se especifica en el parámetro de inicialización **"controladorBaseDeDatos"**). En las líneas 24 y 25 se trata de abrir una conexión a la base de datos **encuestadeanimales**. El nombre de la base de datos se especifica en el parámetro de inicialización **"nombreBaseDeDatos"**. La base de datos contiene una tabla (**resultadosencuesta**) que consiste de tres campos: un entero único para identificar a cada registro (**id**), una cadena que representa la opción de la encuesta (**opcionencuesta**) y un entero que representa el número de votos para la opción de una encuesta (**votos**). [*Nota:* La carpeta de ejemplos para este capítulo contiene una secuencia de comandos de SQL (**encuestadeanimales.sql**) con la cual usted podrá crear la base de datos **encuestadeanimales** para este ejemplo. Para obtener información sobre cómo ejecutar la secuencia de comandos de SQL, consulte el capítulo 23.]

Cuando un usuario envía una respuesta a la encuesta, el método **doPost** (líneas 41 a 127) se encarga de la petición. En las líneas 64 y 65 se obtiene la respuesta a la encuesta y después en las líneas 69 a 115 se trata de procesar esa respuesta. En las líneas 72 y 73 se especifica una consulta para incrementar el valor de **votos** para el registro con el ID especificado y se actualiza la base de datos. En las líneas 78 a 80 se ejecuta la consulta especificada en la línea 77 para recuperar el número total de votos recibidos, utilizando la herramienta de suma integrada en SQL para totalizar todos los votos en la tabla **resultadosencuesta**. Después, en las líneas 85 a 113 se ejecuta la consulta (se devuelven todos los datos en la tabla **resultadosencuesta**) especificada en las líneas 83 y 84, y se procesa el objeto **ResultSet** para crear el resumen de la encuesta para el cliente. Cuando el contenedor de servlets termina la ejecución del servlet, el método ***destroy*** (líneas 130 a 142) cierra el objeto **Statement** y después cierra la conexión a la base de datos. En la figura 24.21 se muestra

http://librosysolucionarios.net


```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig 24.21: Encuesta.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Encuesta</title>
10 </head>
11
12 <body>
13 <form method = "post" action = "/cpe5/encuestadeanimales">
14
15    <p>¿Cuál es su animal favorito?</p>
16
17    <p>
18       <input type = "radio" name = "animal"
19          value = "1" />Perro<br />
20       <input type = "radio" name = "animal"
21          value = "2" />Gato<br />
22       <input type = "radio" name = "animal"
23          value = "3" />Pájaro<br />
24       <input type = "radio" name = "animal"
25          value = "4" />Serpiente<br />
26       <input type = "radio" name = "animal"
27          value = "5" checked = "checked" />Ninguno
28    </p>
29
30    <p><input type = "submit" value = "Enviar" /></p>
31
32 </form>
33 </body>
34 </html>
```

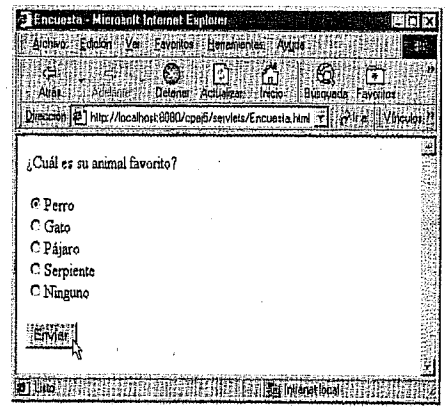


Figura 24.21 Documento **Encuesta.html** que permite a los usuarios enviar las respuestas de la encuesta a **ServletEncuesta**. (Parte 1 de 2.)

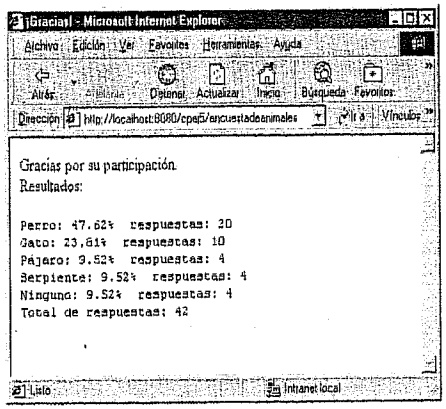


Figura 24.21 Documento `Encuesta.html` que permite a los usuarios enviar las respuestas de la encuesta a `ServletEncuesta`. (Parte 2 de 2.)

el documento `encuesta.html`, el cual invoca a `ServletEncuesta` a través del alias `encuesta-deanimales` cuando el usuario envía el formulario.

Utilizamos nuestro directorio raíz de contexto `cpej5` para demostrar el servlet de la figura 24.20. Coloque a `Encuesta.html` en el directorio `servlets` que creó anteriormente. Coloque a `ServletEncuesta.class` (con la estructura del paquete completa) en el subdirectorio `classes` de `WEB-INF`, en el directorio raíz de contexto `cpej5`. Después edite el descriptor de despliegue `web.xml` en el directorio `WEB-INF` para incluir la información especificada en la figura 24.22. Además, este programa no podrá ejecutarse en Tomcat a

Elemento del descriptor	Valor
elemento `servlet`	
`servlet-name`	`encuestadeanimales`
`description`	`Conexión a una base de datos desde un servlet.`
`servlet-class`	`com.deitel.cpej5.servlets.ServletEncuesta`
`init-param`	
`param-name`	`ubicacionBaseDeDatos`
`param-value`	`C:/CloudScape_5.1`
`init-param`	
`param-name`	`controladorBaseDeDatos`
`param-value`	`com.ibm.db2j.jdbc.DB2jDriver`
`init-param`	
`param-name`	`nombreBaseDeDatos`
`param-value`	`jdbc:db2j:encuestadeanimales`
elemento `servlet-mapping`	
`servlet-name`	`encuestadeanimales`
`url-pattern`	`/encuestadeanimales`

Figura 24.22 Información del descriptor de despliegue para el servlet `ServletEncuesta`.

http://librosysolucionarios.net

menos que la aplicación Web tenga acceso a los archivos JAR que contienen el controlador de bases de datos de Cloudscape y sus clases de soporte. Estos archivos JAR se encuentran en el directorio `lib` de la instalación de Cloudscape en su equipo. Coloque *copias* de estos archivos JAR en el subdirectorio `lib` de `WEBINF` para que estén disponibles para la aplicación Web. Consulte el capítulo 23 si desea obtener más información acerca de cómo configurar Cloudscape.

Después de copiar estos archivos, reinicie Tomcat y escriba el siguiente URL en su navegador Web:

```
http://localhost:8080/cpej5/servlets/Encuesta.html
```

Seleccione un animal y oprima el botón **Enviar** para invocar el servlet.

24.8 Recursos en Internet y World Wide Web

En esta sección se muestra una variedad de recursos disponibles en Internet, junto con una breve descripción de cada uno.

`java.sun.com/products/servlet/index.html`
La página de servlets en el sitio Web de Java de Sun Microsystems, Inc. proporciona acceso a la información más reciente sobre servlets y demás recursos.

`jakarta.apache.org`
Ésta es la página inicial del Proyecto Apache para el *Proyecto Jakarta. Tomcat* (la implementación de referencia de servlets y JavaServerPages) es uno de muchos subproyectos del Proyecto Jakarta.

`jakarta.apache.org/tomcat/index.html`
La página inicial para la implementación de referencia de servlets y JavaServer Pages denominada Tomcat.

`java.apache.org`
Ésta es la página inicial del Proyecto Apache para todas las tecnologías relacionadas con Java. Este sitio proporciona acceso a muchos paquetes de Java que son útiles para los desarrolladores de servlets y JSPs.

`www.servlets.com`
Éste es el sitio Web del libro *Java Servlet Programming* publicado por O'Reilly. El libro proporciona una variedad de recursos. Este libro es un excelente recurso para los programadores que están aprendiendo acerca de los servlets.

`theserverside.com`
TheServerSide.com es un sitio dedicado a mostrar información y recursos para la J2EE.

`www.servletsource.com`
ServletSource.com es un sitio de recursos sobre servlets en general, que contiene código, tips, tutoriales y vínculos hacia muchos otros sitios Web con información sobre servlets.

`www.cookiecentral.com`
Un recurso completo para cookies.

`devedge.netscape.com/library/manuals/2000/javascript/1.3/reference/cookies.html`
Una descripción de las cookies de Netscape.

`www.javacorporate.com/expresso/frame.jsp`
Página inicial del *Marco de trabajo Expresso* de código fuente abierto, el cual incluye una biblioteca de componentes de servlets extensibles para ayudar a agilizar el desarrollo de aplicaciones Web.

`www.servlet.com/srvdev.jhtml`
El foro para desarrolladores de servlets de ServletInc proporciona recursos para los desarrolladores de Java del lado servidor, e información acerca de los servidores Web que soportan las tecnologías de servlets.

`www.servletforum.com`
ServletForum.com es un grupo de noticias en donde usted puede publicar preguntas para que sus colegas las respondan.

`www.coolservlets.com`
Proporciona servlets de Java de código fuente abierto.

`www.getusalinks.org/oo_java_servlets.html`
Proporciona una lista de vínculos hacia recursos sobre servlets y otras tecnologías.

`www.javaskyline.com`
Java Skyline es una revista en línea para desarrolladores de servlets.

`www.rfc-editor.org`
RFC Editor proporciona un motor de búsqueda para RFCs (Peticiones de comentarios). Muchos de estos RFCs proporcionan detalles acerca de las tecnologías relacionadas con Web. Los RFCs de interés para los desarrolladores de servlets son: *URI in WWW* (RFC 1630), *URI: generic syntax* (RFC 2396), *HTTP State Management Mechanism* (RFC 2109), *Use and Interpretation of HTTP Version Numbers* (RFC 2145), *Hyper Text Coffee Pot Control Protocol* (RFC 2324), HTTP/1.1 (RFC 2616) and *HTTP Authentication: Basic and Digest Authentication* (RFC 2617).

RESUMEN

- Las clases e interfaces utilizadas para definir servlets se encuentran en los paquetes `javax.servlet` y `javax.servlet.http`.
- Internet ofrece muchos protocolos. El protocolo HTTP (Protocolo de transferencia de hipertexto), que forma la base de World Wide Web, utiliza URLs (Localizadores uniformes de recursos) para localizar recursos en Internet.
- Los URLs representan archivos o directorios y pueden representar tareas complejas como búsquedas en bases de datos y en Internet.
- La tecnología JavaServer Pages es una extensión de la tecnología de los servlets.
- Los servlets son ejecutados normalmente por el componente contenedor de servlets de un servidor de aplicaciones Web.
- Los servlets y las JavaServer Pages se han vuelto tan populares que ahora se soportan en la mayoría de los principales servidores Web y de aplicaciones.
- Todos los servlets deben implementar a la interfaz `Servlet`. Los métodos de esta interfaz son invocados automáticamente por el contenedor de servlets.
- El ciclo de vida de un servlet empieza cuando el contenedor de servlets lo carga en memoria; generalmente, en respuesta a la primera petición que recibe ese servlet. Antes de que el servlet pueda encargarse de la primera petición, el contenedor de servlets invoca al método `init` de ese servlet. Una vez que `init` termina de ejecutarse, el servlet puede responder a su primera petición. Todas las peticiones son manejadas por el método `service` de un servlet, el cual puede ser llamado muchas veces durante el ciclo de vida de un servlet. Cuando el contenedor de servlets termina el servlet, se hace una llamada al método `destroy` de ese servlet para liberar sus recursos.
- Los paquetes de servlets definen dos clases `abstract` que implementan a la interfaz `Servlet`: las clases `GenericServlet` y `HttpServlet`. La mayoría de los servlets extienden a una de estas clases y sobrescriben a algunos o a todos sus métodos, con comportamientos personalizados apropiadamente.
- El método clave en cada servlet es `service`, el cual recibe tanto a un objeto `ServletRequest` como a un objeto `ServletResponse`. Estos objetos proporcionan acceso a los flujos de entrada y salida que permiten al servlet leer datos del cliente y enviar datos al cliente.
- Los servlets basados en Web generalmente extienden a la clase `HttpServlet`. La clase `HttpServlet` sobrescribe el método `service` para diferenciar entre las peticiones típicas recibidas de un navegador Web cliente. Los dos tipos de peticiones de HTTP más comunes (también conocidos como métodos de petición) son `get` y `post`.
- La clase `HttpServlet` define los métodos `doGet` y `doPost` para responder a las peticiones `get` y `post` de un cliente, respectivamente. Estos métodos se llaman mediante el método `service` de la clase `HttpServlet`, el cual se llama cuando llega una petición al servidor.
- Los métodos `doGet` y `doPost` reciben como argumentos un objeto `HttpServletRequest` y un objeto `HttpServletResponse`, los cuales permiten la interacción entre el cliente y el servidor.
- Una respuesta se envía al cliente a través de un objeto `PrintWriter` devuelto por el método `getWriter` del objeto `HttpServletResponse`.
- El método `setContentType` del objeto `HttpServletResponse` especifica el tipo MIME de la respuesta al cliente. Esto permite al navegador cliente comprender y manejar el contenido.
- El servidor `localhost` (dirección IP `127.0.0.1`) es un nombre de host reconocido en los equipos que soportan protocolos de red basados en TCP/IP, como HTTP. Este nombre de host puede usarse para probar aplicaciones de TCP/IP en el equipo local.
- El servidor Tomcat espera las peticiones de los clientes en el puerto 8080. Este número de puerto debe especificarse como parte del URL para pedir un servlet que se ejecute en Tomcat.

- El cliente puede acceder a un servlet solamente si éste se encuentra instalado en un servidor que pueda responder a las peticiones de servlets. Los servidores Web que soportan servlets generalmente tienen un procedimiento de instalación para los servlets.
- Tomcat es una implementación completamente funcional de los estándares de JSP y servlets. Incluye un servidor Web, por lo que puede usarse como un contenedor de prueba independiente para JSPs y servlets.
- Tomcat puede especificarse como el manejador para las peticiones de JSPs y servlets recibidas por los servidores Web populares como Apache e IIS. Tomcat está integrado también en la implementación de referencia Java 2 Enterprise Edition de Sun Microsystems.
- Las JSPs, los servlets y sus archivos de soporte se despliegan como parte de aplicaciones Web. En Tomcat, las aplicaciones Web se despliegan en el subdirectorio `webapps` de la instalación de Tomcat.
- Una aplicación Web tiene una estructura de directorios reconocida, en la que residen todos los archivos que forman parte de la aplicación. Esta estructura de directorios puede ser creada por el administrador del servidor Tomcat en el directorio `webapps`, o toda la estructura de directorios completa puede archivarse en un archivo de fichero de aplicación Web. Dicho archivo se conoce como archivo WAR y termina con la extensión de archivo `.war`.
- Si un archivo WAR se coloca en el directorio `webapps`, entonces cuando el servidor Tomcat empiece a ejecutarse, extraerá el contenido del archivo WAR en la estructura de subdirectorios apropiada de `webapps`.
- La estructura de directorios de aplicaciones Web se separa en un directorio raíz de contexto (el directorio de nivel superior para toda una aplicación Web) y varios subdirectorios. El directorio raíz de contexto es el directorio raíz para la aplicación Web. Todas las JSPS, documentos de HTML, servlets y archivos de soporte como imágenes y archivos de clases residen en este directorio o sus subdirectorios. El directorio `WEB-INF` contiene el descriptor de despliegue de la aplicación Web (`web.xml`). El directorio `WEB-INF/classes` contiene los archivos de clases del servlet y demás archivos de clases de soporte que se utilizan en una aplicación Web. El directorio `WEB-INF/lib` contiene archivos de ficheros de Java (JAR), los cuales pueden incluir archivos de clases de Java y demás archivos de soporte utilizados en una aplicación Web.
- Antes de desplegar una aplicación Web, el contenedor de servlets debe conocer el directorio raíz de contexto de la aplicación Web. En Tomcat sólo se necesita colocar un directorio en el subdirectorio `webapps`. Tomcat utiliza el nombre de ese directorio como el nombre de contexto.
- Para desplegar una aplicación Web se requiere la creación de un descriptor de despliegue (`web.xml`).
- Las peticiones `get` de HTTP pueden escribirse directamente en el campo de dirección de su navegador.
- En una petición `get`, los parámetros se pasan como pares nombre/valor. Un `?` separa el URL de los datos que se pasan como parte de una petición `get`. Los pares nombre/valor se pasan con el nombre y el valor separados por un `=`. Si hay más de un par nombre/valor, cada par se separa por un `&`.
- El método `getParameter` de la interfaz `HttpServletRequest` recibe el nombre del parámetro como un argumento y devuelve el valor `String` correspondiente, o `null` si el parámetro no forma parte de la petición.
- Una petición `post` de HTTP se utiliza comúnmente para colocar los datos de un formulario de página Web en un manejador de formularios del lado servidor que procesa los datos.
- Los navegadores a menudo colocan en caché (almacenan en disco) las páginas Web, para poder volver a cargarlas rápidamente. Los navegadores no colocan en caché la respuesta del servidor a una petición `post`.
- El método `doPost` recibe los mismos dos argumentos que `doGet`: un objeto que implementa a la interfaz `HttpServletRequest` para representar la petición del cliente y un objeto que implementa a la interfaz `HttpServletResponse` para representar la respuesta del servlet.
- El método `sendRedirect` de `HttpServletResponse` redirecciona una petición hacia el URL especificado.
- Cuando un servlet utiliza una ruta relativa para hacer referencia a otro recurso estático o dinámico, el servlet asume que se va a utilizar el mismo directorio raíz de contexto, a menos que se especifique un URL completo para el recurso.
- Una vez que se ejecuta el método `sendRedirect`, el servlet que llamó a `sendRedirect` termina el procesamiento de la petición.
- Al redireccionar peticiones, los parámetros de la petición original se pasan como parámetros a la nueva petición. También pueden pasarse parámetros de petición adicionales.
- Los nuevos parámetros se agregan a los ya existentes. Si un nuevo parámetro tiene el mismo nombre que uno existente, el nuevo parámetro tiene precedencia sobre el valor original. Sin embargo, todos los valores se pasan de todas formas.
- El conjunto completo de valores para un nombre de parámetro de petición dado puede obtenerse llamando al método `getParameterValues` de la interfaz `HttpServletRequest`, el cual recibe el nombre del parámetro como un argumento y devuelve un arreglo de objetos `String` que contienen los valores del parámetro, en orden del valor más recientemente agregado al menos reciente.

http://librosysolucionarios.net

- Muchas de las aplicaciones de la actualidad son aplicaciones distribuidas de tres niveles, las cuales consisten en una interfaz de usuario, la lógica de negocios y una base de datos.
- En arquitecturas de multinivel, a menudo se utilizan servidores Web en el nivel intermedio. Los componentes del lado servidor (como los servlets) se ejecutan en un servidor de aplicaciones junto con el servidor Web. Estos componentes proporcionan la lógica de negocios que manipula la información de las bases de datos y que se comunica con los navegadores Web clientes.
- El método `init` de `Servlet` toma un argumento `ServletConfig` y lanza una excepción `ServletException`. El argumento proporciona al servlet la información acerca de sus parámetros de inicialización, los cuales se especifican en un elemento `servlet` en el descriptor de despliegue. Cada parámetro aparece en un elemento `init-param` con los elementos hijos `param-name` y `param-value`.

TERMINOLOGÍA

aplicación Web
asociación de servlet
`CATALINA_HOME`, variable de entorno
ciclo de vida de un servlet
clientes delgados
colocar en caché una página Web
contenedor de servlets
descriptor (`web.xml`)
descriptor de despliegue
desplegar una aplicación Web
despliegue de aplicaciones Web
`destroy`, método de `Servlet`
`doGet`, método de `HttpServlet`
`doPost`, método de `HttpServlet`
encabezado de HTTP
`GenericServlet`, clase de `javax.servlet`
`get`, petición
`getCookies`, método de `HttpServletRequest`
`getOutputStream`, método de `HTTPServletResponse`
`getParameter`, método de `HttpServletRequest`
`getParameterNames`, método de `HttpServletRequest`
`getParameterValues`, método de `HttpServletRequest`
`getWriter`, método de `HTTPServletResponse`
HTTP (Protocolo de transferencia de hipertexto)
`HttpServlet`, interfaz
`HttpServletRequest`, interfaz
`HttpServletResponse`, interfaz
`init`, método de `Servlet`
`JAVA_HOME`, variable de entorno
`javax.servlet`, paquete
`javax.servlet.http`, paquete
`localhost ( 127.0.0.1)`
método de petición
nombre de host
número de puerto reconocido
parámetro de inicialización
parámetro de petición
patrón de URL
petición de HTTP
`post`, petición
proyecto Jakarta
puerto
`put`, petición
raíz de contexto
redireccionar una petición
`sendRedirect`, método de `HttpServletResponse`
`service`, método de `Servlet`
servlet
`Servlet`, interfaz
`ServletException`, clase
`ServletOutputStream`, clase
`ServletRequest`, interfaz
`ServletResponse`, interfaz
`setContentType`, método de `HttpServletResponse`
servidor Apache Tomcat
`text/html`, tipo MIME
tipo MIME
`trace`, petición
WAR (archivo de aplicación Web), archivo
`webapps`, directorio
`WEB-INF`, directorio
`WEB-INF/classes`, directorio
`WEB-INF/lib`, directorio

EJERCICIOS DE AUTOEVALUACIÓN

24.1 Complete las siguientes oraciones:
a) Las clases `HttpServlet` y `GenericServlet` implementan a la interfaz _____.
b) La clase `HttpServlet` define los métodos _____ y _____ para responder a las peticiones `get` y `post` de un cliente.
c) El método _____ de `HttpServletResponse` obtiene un flujo de salida basado en caracteres que permite enviar datos de texto al cliente.
d) El atributo _____ de `form` especifica el *manejador de formularios* del lado servidor (es decir, el programa que maneja la petición).
e) _____ es el nombre de host reconocido que se refiere a su propio equipo.

24.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.
a) Los servlets se utilizan generalmente en el lado cliente de una aplicación de red.
b) Los métodos de los servlets son ejecutados por el contenedor de servlets.

http://librosysolucionarios.net

c) Las dos peticiones de HTTP más comunes son `get` y `put`.
d) El número de puerto reconocido para las peticiones Web es 55.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

24.1 a) `Servlet`. b) `doGet`, `doPost`. c) `getWriter`. d) `action`. e) `localhost`.

24.2 a) Falso. Los servlets se utilizan generalmente en el lado del servidor.
b) Verdadero.
c) Falso. Las dos peticiones de HTTP más comunes son `get` y `post`.
d) Falso. El número de puerto reconocido para las peticiones Web es 80.

EJERCICIOS

24.3 Cree una aplicación Web para FAQs dinámicas. La aplicación deberá obtener la información para crear la página Web de FAQs dinámicas de una base de datos que consista de una tabla `Temas` y una tabla `FAQ`. La tabla `Temas` deberá tener dos campos: un número de identificación (ID) entero único para cada tema (`idTema`) y un nombre para cada tema (`nombreTema`). La tabla `FAQ` deberá tener tres campos: el `idTema` (una clave externa), una cadena que represente la pregunta (`pregunta`) y la respuesta a esa pregunta (`respuesta`). Al invocar el servlet, éste deberá leer la información de la base de datos y deberá devolver una página Web creada en forma dinámica, que contenga cada pregunta y respuesta ordenadas por tema.

24.4 Modifique la aplicación Web del ejercicio 24.3 de manera que la petición inicial al servlet devuelva una página Web de temas en la base de datos de FAQs. Así, el usuario podrá hacer clic en el hipervínculo hacia otro servlet que devuelva solamente las preguntas frecuentes para un tema específico.

24.5 Modifique la aplicación Web de la figura 24.20 para permitir al usuario ver los resultados de la encuesta sin tener que responder a la misma.

24.6 Modifique la aplicación Web de la figura 24.20 para poder utilizarla en forma genérica con cualquier encuesta de la forma apropiada. Use parámetros de servlet (como se describió en la sección 24.7) para especificar las opciones de la encuesta. Cuando el usuario pida la encuesta, genere dinámicamente un elemento `form` que contenga las opciones de la encuesta. Despliegue esta aplicación Web dos veces, utilizando directorios raíz de contexto distintos. *Nota:* Tal vez necesite modificar la base de datos en este ejemplo, de manera que pueda almacenar varias encuetas a la vez.

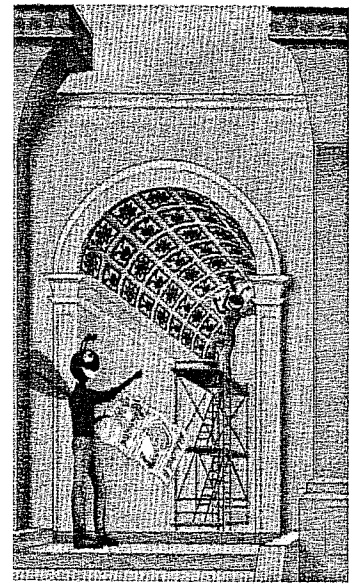
24.7 Escriba una aplicación Web que consista de un servlet (`ServletDirectorio`) y varios documentos Web. El documento `index.html` debe ser el primer documento que vea el usuario. En ese documento usted deberá tener una serie de hipervínculos hacia otras páginas en su sitio Web. Al hacer clic en cada hipervínculo, éste deberá invocar el servlet con una petición `get` que contenga un parámetro `pagina`. El servlet deberá obtener el parámetro `pagina` y redireccionar la petición hacia el documento apropiado.

25

JavaServer Pages (JSP)

Objetivos

- Crear y desplegar JavaServer Pages.
- Utilizar scriptlets y objetos implícitos de las JSPs para crear páginas Web dinámicas.
- Especificar información global de JSPs mediante directivas.
- Utilizar acciones para manipular JavaBeans en una JSP, para incluir recursos en forma dinámica y reenviar peticiones hacia otras JSPs.



Un tomate no se comunica con otro tomate, es lo que creemos. Podríamos estar equivocados.
Gustav Eckstein

Un burro me parece un caballo traducido al holandés.
Georg Christoph Licthtenberg

El talento es cuestión de cantidad. El talento no escribe una página: escribe trescientas.
Jules Renard

Cada acción se debe a una u otra de siete causas: probabilidad, naturaleza, compulsión, hábito, razonamiento, ira o apetito.
Aristóteles

http://librosysolucionarios.net

Plan general

25.1 Introducción
25.2 Generalidades acerca de las JavaServer Pages
25.3 Nuestro primer ejemplo de JavaServer Page
25.4 Objetos implícitos
25.5 Secuencias de comandos
25.5.1 Componentes de las secuencias de comandos
25.5.2 Ejemplo de secuencia de comandos
25.6 Acciones estándar
25.6.1 La acción `<jsp:include>`
25.6.2 La acción `<jsp:forward>`
25.6.3 La acción `<jsp:useBean>`
25.7 Directivas
25.7.1 La directiva `page`
25.7.2 La directiva `include`
25.8 Ejemplo práctico: Libro de visitantes
25.9 Recursos en Internet y World Wide Web
Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

25.1 Introducción

En este capítulo continuaremos nuestra discusión sobre las redes cliente-servidor con la tecnología *JavaServer Pages (JSP)*: una extensión de la tecnología de los servlets. Las JavaServer Pages simplifican la producción de contenido Web dinámico. Permiten a los programadores de aplicaciones Web crear contenido dinámico mediante la reutilización de componentes predefinidos, y mediante la interacción con los componentes utilizando secuencias de comandos del lado del servidor. Los programadores de una JavaServer Page pueden reutilizar JavaBeans y crear bibliotecas de marcas personalizadas que encapsulen una funcionalidad compleja y dinámica. Las bibliotecas de marcas personalizadas permiten incluso a los diseñadores Web que no están familiarizados con Java mejorar las páginas Web con poderosas herramientas de procesamiento y contenido dinámico.

Además de los tipos para la programación de servlets (capítulo 24), las clases e interfaces específicas para la programación con JavaServer Pages se encuentran en los paquetes `javax.servlet.jsp` y `javax.servlet.jsp.tagext`. Hablaremos sobre muchas de estas clases e interfaces a lo largo de este capítulo, a medida que presentemos los fundamentos sobre las JavaServer Pages. Para obtener una descripción completa de las JavaServer Pages, consulte la especificación JavaServer Pages 1.2, que puede descargarse desde la página `java.sun.com/products/jsp/download.html`. También incluimos otros recursos sobre JSPs en la sección 25.9. (*Nota:* El código fuente y las imágenes de los ejemplos de este capítulo podrá encontrarlos en el CD que acompaña a este libro y en `www.deitel.com`.)

25.2 Generalidades acerca de las JavaServer Pages

Hay cuatro componentes clave de las JSPs: *directivas*, *acciones*, *scriptlets* y *bibliotecas de marcas*. Las directivas son mensajes para el contenedor de JSPs, que permiten al programador especificar configuraciones de página, incluir contenido de otros recursos y especificar bibliotecas de marcas personalizadas para usarlas en una JSP. Las acciones encapsulan la funcionalidad en marcas predefinidas que los programadores pueden incrustar en una JSP. A menudo las acciones se realizan con base en la información que se envíe al servidor como parte de una petición de cliente específica. También pueden crear objetos de Java para utilizarlos en scriptlets de JSP.

Los scriptlets, o *elementos de secuencias de comandos*, permiten a los programadores insertar código de Java que interactúe con los componentes de una JSP (y posiblemente con otros componentes de la aplicación Web) para realizar el procesamiento de las peticiones. Las bibliotecas de marcas son parte del *mecanismo de extensión de marcas* que permite a los programadores crear marcas personalizadas. Dichas marcas permiten a los programadores manipular el contenido de una JSP. Estos tipos de componentes de JSPs se describirán con detalle en las siguientes secciones.

En ciertas formas, las JavaServer Pages se ven como un documento estándar de XHTML o XML. De hecho, las JSPs generalmente incluyen marcado de XHTML o XML. A dicho marcado se le conoce como *datos de plantilla fija* o *texto de plantilla fija*. Los datos de plantilla fija a menudo ayudan a que un programador decida si va a utilizar un servlet o una JSP. Los programadores tienden a utilizar JSPs cuando la mayoría del contenido que se envía al cliente está compuesto por datos de plantilla fija y sólo una pequeña porción del contenido se genera en forma dinámica mediante código de Java. Los programadores comúnmente utilizan servlets cuando sólo una pequeña porción del contenido que se envía al cliente está compuesta por datos de plantilla fija. De hecho, algunos servlets no producen contenido. En vez de ello, realizan una tarea a beneficio del cliente y después invocan a otros servlets o JSPs para que proporcionen una respuesta. Observe que, en la mayoría de los casos, las tecnologías de servlets y JSPs son intercambiables. Al igual que los servlets, las JSPs por lo general se ejecutan como parte de un servidor Web. El componente de servidor que las ejecuta se conoce comúnmente como el *contenedor de JSPs*.

Observación de ingeniería de software 25.1

El texto literal en una JSP se convierte en literales de cadena en el servlet que representa a la JSP traducida.

Cuando un servidor habilitado para usar JSPs recibe la primera petición de una JSP, el contenedor de JSPs traduce esa JSP en un servlet de Java que maneja la petición actual y las peticiones futuras de esa JSP. Si hay errores al compilar el nuevo servlet, estos errores se convierten en *errores en tiempo de traducción*. El contenedor de JSPs coloca las instrucciones de Java que implementan la petición de la JSP en el método `_jspService`, en tiempo de traducción. Si el nuevo servlet se compila apropiadamente, el contenedor de JSPs invoca al método `_jspService` para procesar la petición. La JSP puede responder directamente a la petición, o puede invocar a otros componentes de la aplicación Web para que le ayuden a procesar la petición. Cualquier error que ocurra durante el procesamiento de la petición se denomina *error en tiempo de petición*.

Tip de rendimiento 25.1

Algunos contenedores de JSPs traducen las JSPs a servlets en tiempo de instalación. Esto elimina la sobrecarga de la traducción para el primer cliente que pide a cada JSP.

En general, el mecanismo de petición/respuesta y el ciclo de vida de una JSP son iguales que los de un servlet. Las JSPs pueden definir los métodos `jspInit` y `jspDestroy` (similares a los métodos de servlet `init` y `destroy`), que son invocados por el contenedor de JSPs al inicializar y terminar una JSP, respectivamente. Los programadores de JSPs pueden definir estos métodos utilizando *declaraciones* de JSP (parte del mecanismo de secuencia de comandos de JSP).

25.3 Nuestro primer ejemplo de JavaServer Page

Empezaremos nuestra introducción a las JavaServer Pages con un ejemplo simple (figura 25.1) en el que se insertan la fecha y hora actuales en una página Web, mediante el uso de una expresión de JSP.

Como puede ver, la mayor parte de `reloj.jsp` consiste en marcado de XHTML. En casos como éste, las JSPs son más fáciles de implementar que los servlets. En un servlet que realice la misma tarea que esta JSP, cada línea de marcado de XHTML viene siendo por lo general una instrucción de Java separada, la cual envía la cadena que representa el marcado como parte de la respuesta al cliente. Comúnmente, al escribir código para mostrar marcado se pueden producir errores. La mayoría de los editores de JSPs proporcionan la capacidad de colorear la sintaxis, para ayudar a los programadores a verificar que el marcado tenga una sintaxis apropiada.

Observación de ingeniería de software 25.2

Las JavaServer Pages son más fáciles de implementar que los servlets cuando la respuesta a una petición de un cliente consiste principalmente de marcado que permanece constante entre cada petición.

http://librosysolucionarios.net

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.1: reloj.jsp -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9     <head>
10       <meta http-equiv = "refresh" content = "60" />
11
12       <title>Un ejemplo simple de JSP</title>
13
14       <style type = "text/css">
15          .grande { font-family: helvetica, arial, sans-serif;
16                    font-weight: bold;
17                    font-size: 2em; }
18       </style>
19    </head>
20
21    <body>
22       <p class = "grande">Ejemplo simple de JSP</p>
23
24       <table style = "border: 6px outset;">
25          <tr>
26             <td style = "background-color: black;">
27                <p class = "grande" style = "color: cyan;">
28
29                   <!-- expresión de JSP para insertar la fecha/hora -->
30                   <%= new java.util.Date() %>
31
32                </p>
33             </td>
34          </tr>
35       </table>
36    </body>
37
38 </html>
```

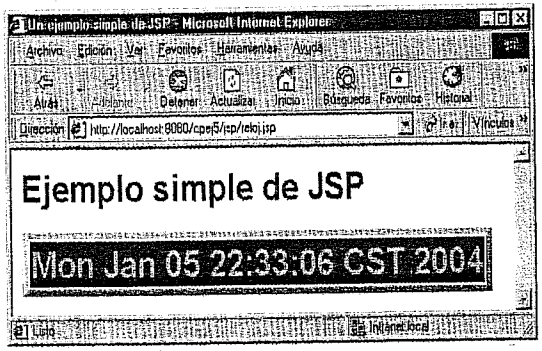


Figura 25.1 Expresión de JSP para insertar la fecha y hora en una página Web. (Parte 1 de 2.)

http://librosysolucionarios.net

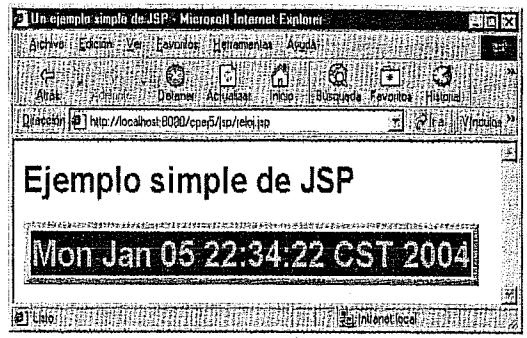


Figura 25.1 Expresión de JSP para insertar la fecha y hora en una página Web. (Parte 2 de 2.)

La JSP de la figura 25.1 genera un documento de XHTML que muestra la fecha y hora actuales. La línea clave en esta JSP (línea 30) es la expresión:

```
<%= new java.util.Date() %>
```

Las expresiones de JSP están delimitadas por `<%=` y `%>`. La expresión anterior crea una nueva instancia de la clase `Date` (paquete `java.util`). Cuando el cliente pide esta JSP, la expresión anterior inserta la representación `String` de la fecha y hora en la respuesta al cliente. [*Nota:* Para tener una internacionalización apropiada, se requiere que la JSP devuelva la fecha en el formato de la configuración regional del cliente. En este ejemplo, la configuración regional del servidor determina la representación `String` de la fecha. En la figura 25.9, `reloj2.jsp` demostrará cómo determinar la configuración regional del cliente y utilizará un objeto `DateFormat` (paquete `java.text`) para dar formato a la fecha, utilizando esa configuración regional.

Observación de ingeniería de software 25.3

El contenedor de JSPs convierte el resultado de cada expresión de JSP en una cadena que se envía como parte de la respuesta al cliente.

Utilizamos el *elemento* `meta` de XHTML en la línea 10 para establecer un *intervalo de actualización* de 60 segundos para el documento. Esto hace que el navegador pida la página `reloj.jsp` cada 60 segundos. Para cada petición de `reloj.jsp`, el contenedor de JSPs vuelve a evaluar la expresión de la línea 30, creando un nuevo objeto `Date` con la fecha y hora actuales del servidor.

Al igual que en el capítulo 24, utilizaremos Apache Tomcat para probar nuestras JSPs en la aplicación Web `cpej5` que creamos anteriormente. Para obtener detalles acerca de cómo crear y configurar la aplicación Web `cpej5`, consulte las secciones 24.3.1 y 24.3.2. Para probar `reloj.jsp`, cree un nuevo directorio llamado `jsp` en el subdirectorio `cpej5` del directorio `webapps` de Tomcat. A continuación, copie el archivo `reloj.jsp` en el directorio `jsp`. Abra su navegador Web y escriba el siguiente URL para probar la página `reloj.jsp`:

```
http://localhost:8080/cpej5/jsp/reloj.jsp
```

La primera vez que invoque la JSP, observará un ligero retraso a medida que Tomcat traduce la JSP en un servlet e invoca a este servlet para que responda a su petición. (*Nota:* No es necesario crear un directorio llamado `jsp` en una aplicación Web. Utilizamos este directorio para separar los ejemplos en este capítulo de los ejemplos de servlets en el capítulo 24.)

25.4 Objetos implícitos

Los *objetos implícitos* proporcionan a los programadores el acceso a muchas herramientas de servlets, dentro del contexto de una JavaServer Page. Los objetos implícitos tienen cuatro alcances: *aplicación*, *página*, *petición* y *sesión*. La JSP y la aplicación contenedora de servlets poseen objetos con *alcance de aplicación*. Cual-

quier servlet o JSP puede manipular a dichos objetos. Los objetos con *alcance de página* existen solamente en la página que los define. Cada página tiene sus propias instancias de los objetos implícitos con alcance de página. Los objetos con *alcance de petición* existen durante el tiempo de vida de la petición. Por ejemplo, una JSP puede procesar parcialmente una petición y después reenviarla hacia otro servlet o JSP para seguir procesándola. Los objetos con alcance de petición quedan fuera de alcance cuando se completa el procesamiento de la petición mediante una respuesta al cliente. Los objetos con *alcance de sesión* existen durante toda la sesión de navegación del cliente. En la figura 25.2 se describen los objetos implícitos de JSP y sus alcances. En este capítulo demostraremos el uso de varios de estos objetos.

Observe que muchos de los objetos implícitos extienden clases o implementan interfaces que describimos en el capítulo 24. Por lo tanto, las JSPs pueden utilizar los mismos métodos que los servlets para interactuar con dichos objetos, según lo descrito en el capítulo 24. La mayoría de los ejemplos en este capítulo utilizan uno o más de los objetos implícitos de la figura 25.2.

25.5 Secuencias de comandos

Las JavaServer Pages a menudo presentan el contenido generado en forma dinámica como parte de un documento de XHTML que se envía al cliente en respuesta a una petición. En algunos casos el contenido es estático,

Objeto implícito	Descripción
Alcance de aplicación	
`application`	Este objeto `javax.servlet.ServletContext` representa el contenedor en el que se ejecuta la JSP.
Alcance de página	
`config`	Este objeto `javax.servlet.ServletConfig` representa las opciones de configuración de la JSP. Al igual que con los servlets, las opciones de configuración pueden especificarse en un descriptor de aplicación Web.
`exception`	Este objeto `java.lang.Throwable` representa la excepción que se pasa a la página de error de JSP. Este objeto está disponible solamente en una página de error de JSP.
`out`	Este objeto `javax.servlet.jsp.JspWriter` escribe texto como parte de la respuesta a una petición. Este objeto se utiliza en forma implícita con extensiones y acciones de JSP para insertar contenido de cadena en una respuesta.
`page`	Este objeto `java.lang.Object` representa la referencia `this` para la instancia actual de la JSP.
`pageContext`	Este objeto `javax.servlet.jsp.PageContext` oculta los detalles de implementación del servlet y contenedor de JSPs subyacentes, y proporciona a los programadores de JSPs el acceso a los objetos implícitos que se describen en esta tabla.
`response`	Este objeto representa la respuesta al cliente y generalmente es una instancia de una clase que implementa a `HttpServletResponse` (paquete `javax.servlet.http`). Si se utiliza un protocolo distinto de HTTP, este objeto es una instancia de una clase que implementa a `javax.servlet.ServletResponse`.
Alcance de petición	
`request`	Este objeto representa la petición del cliente. El objeto normalmente es una instancia de una clase que implementa a `HttpServletRequest` (paquete `javax.servlet.http`) Si se utiliza un protocolo distinto de HTTP, este objeto es una instancia de una clase que implementa a `javax.servlet.ServletRequest`.
Alcance de sesión	
`session`	Este objeto `javax.servlet.http.HttpSession` representa la información de sesión del cliente, si es que se ha creado dicha sesión. Este objeto está disponible solamente en páginas que participen en una sesión.

Figura 25.2 Objetos implícitos de JSP.

http://librosysolucionarios.net

pero se envía solamente si se cumplen ciertas condiciones durante una petición (como el proporcionar valores en un elemento `form` que envía una petición). Los programadores de JSPs pueden insertar código y lógica de Java en una JSP mediante el uso de secuencias de comandos.

25.5.1 Componentes de las secuencias de comandos

Los componentes de las secuencias de comandos de JSP incluyen scriptlets, comentarios, expresiones, declaraciones y secuencias de escape. En esta sección describiremos a cada uno de estos componentes de secuencias de comandos. Muchos de ellos se muestran en la figura 25.4, al final de la sección 25.5.2.

Los scriptlets son bloques de código delimitado por los caracteres `<%` y `%>`. Contienen instrucciones de Java que el contenedor coloca en el método `_jspService` al momento de la traducción.

Las JSPs soportan tres estilos de comentarios: Comentarios de JSP, comentarios de XHTML y comentarios de lenguaje de secuencias de comandos. Los *comentarios de JSP* están delimitados por los caracteres `<%--` y `--%>`. Éstos pueden colocarse en cualquier parte de una JSP, pero no dentro de los scriptlets. Los *comentarios de XHTML* están delimitados por los caracteres `<!--` y `-->`. Estos comentarios pueden colocarse en cualquier parte de una JSP, pero no dentro de los scriptlets. Los comentarios de lenguaje de secuencia de comandos son en realidad comentarios de Java, ya que éste es el único lenguaje de secuencia de comandos para JSP actualmente. Los scriptlets pueden utilizar comentarios de fin de línea de Java (`//`) y los comentarios tradicionales (delimitados por `/*` y `*/`). Los comentarios de JSP y los comentarios de lenguaje de secuencias de comandos se ignoran y no aparecen en la respuesta a un cliente. Cuando los clientes ven el código fuente de una respuesta JSP, sólo ven los comentarios de XHTML en el código fuente. Los distintos estilos de comentarios son útiles para separar los comentarios que el usuario pueda ver de los comentarios que documentan la lógica procesada en el servidor.



Error común de programación 25.1

Si se coloca un comentario de JSP o de XHTML dentro de un scriptlet, se produce un error de sintaxis en tiempo de traducción, el cual evita que la JSP se traduzca apropiadamente.

Una expresión de JSP delimitada por los caracteres `<%=` y `%>` contiene una expresión de Java que se evalúa cuando un cliente pide la JSP que contiene esa expresión. El contenedor convierte el resultado de una expresión JSP en un objeto `String` y después envía ese objeto `String` como parte de la respuesta al cliente.

Las declaraciones (delimitadas por los caracteres `<%!` y `%>`) permiten a un programador de JSPs definir las variables y los métodos a utilizar en una JSP. Las variables se convierten en variables de instancia de la clase de servlet que representa a la JSP traducida. De manera similar, los métodos se convierten en miembros de la clase que representa a la JSP traducida. Las declaraciones de las variables y los métodos en una JSP utilizan la sintaxis de Java. Por lo tanto, la declaración de una variable debe terminar en un punto y coma, como en:

```
<%! int contador = 0; %>
```



Error común de programación 25.2

Declarar una variable sin utilizar un punto y coma de terminación es un error de sintaxis.



Observación de ingeniería de software 25.4

Las JSPs no deben almacenar información de estado del cliente en variables de instancia. En vez de ello, las JSPs deben utilizar el objeto implícito `session`.

Los caracteres o secuencias de caracteres especiales que utiliza normalmente el contenedor de JSPs para delimitar código de JSP pueden incluirse en una JSP como caracteres literales en elementos de secuencias de comandos, datos de plantilla fija y valores de atributos, mediante el uso de *secuencias de escape*. En la figura 25.3 se muestra el carácter o caracteres literales y sus correspondientes secuencias de escape, además de indicar en dónde se pueden utilizar estas secuencias de escape.

25.5.2 Ejemplo de secuencia de comandos

La JSP de la figura 25.4 muestra cómo responder a las peticiones `get` mediante las herramientas básicas de secuencias de comandos. La JSP permite al usuario escribir un nombre de pila y después muestra ese nombre en

Literal	Secuencia de escape	Descripción
<%	<\%	La secuencia de caracteres <% normalmente indica el comienzo de un scriptlet. La secuencia de escape <\% coloca los caracteres literales <% en la respuesta al cliente.
%>	%\>	La secuencia de caracteres %> normalmente indica el final de un scriptlet. La secuencia de escape %\> coloca los caracteres literales %> en la respuesta al cliente.
'	\'	Al igual que con las literales de cadena en un programa de Java, las secuencias de escape para los caracteres ', " y \ permiten que estos caracteres aparezcan en los valores de los atributos. Recuerde que el texto literal en una JSP se convierte en literales de cadena en el servlet que representa a la JSP traducida.
"	\"	
\	\\	

Figura 25.3 Secuencias de escape de JSP.

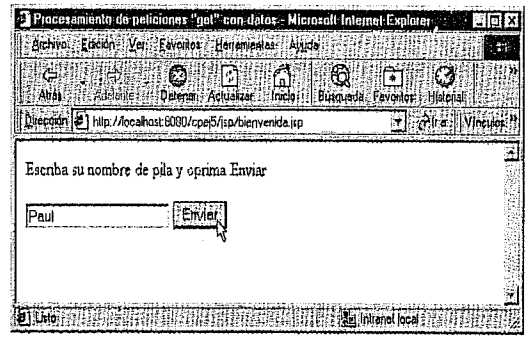
la respuesta. Mediante el uso de secuencias de comandos, la JSP determina si se pasó un parámetro `nombrePila` como parte de la petición; de no ser así, la JSP devuelve un documento de XHTML que contiene un elemento `form`, a través del cual el usuario puede escribir un nombre de pila. En caso contrario, la JSP obtiene el valor de `nombrePila` y lo utiliza como parte de un documento de XHTML que da la bienvenida al usuario al mundo de las JavaServer Pages.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.4: bienvenida.jsp -->
6  <!-- JSP que procesa una petición "get" con datos. -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9
10    <!-- sección de encabezado del documento -->
11    <head>
12       <title>Procesamiento de peticiones "get" con datos</title>
13    </head>
14
15    <!-- sección del cuerpo del documento -->
16    <body>
17       <% // iniciar scriptlet
18
19          String nombre = request.getParameter( "nombrePila" );
20
21          if ( nombre != null ) {
22
23       %> <!-- terminar scriptlet para insertar datos de plantilla fija -->
24
25             <h1>
26                Hola <%= nombre %>, <br />
27                ¡Bienvenido a las JavaServer Pages!
28             </h1>
```

Figura 25.4 Uso de secuencias de comandos en una JavaServer Page: `bienvenida.jsp`. (Parte 1 de 2.)

http://librosysolucionario.net


```
29
30          <% // continuar scriptlet
31
32          } // fin de instrucción if
33          else {
34
35       %> <!-- terminar scriptlet para insertar datos de plantilla fija -->
36
37             <form action = "bienvenida.jsp" method = "get">
38                <p>Escriba su nombre de pila y oprima Enviar</p>
39
40                <p><input type = "text" name = "nombrePila" />
41                   <input type = "submit" value = "Enviar" />
42                </p>
43             </form>
44
45          <% // continuar scriptlet
46
47          } // fin de instrucción else
48
49       %> <!-- fin de scriptlet -->
50    </body>
51
52 </html>  <!-- fin del documento de XHTML -->
```



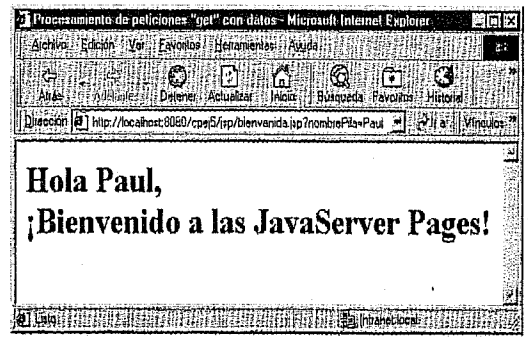


Figura 25.4 Uso de secuencias de comandos en una JavaServer Page: `bienvenida.jsp`. (Parte 2 de 2.)

Observe que la mayor parte del código de la figura 25.4 es marcado de XHTML (es decir, datos de plantilla fija). A lo largo del elemento `body` hay varios scriptlets (líneas 17 a 23, 30 a 35 y 45 a 49) y una expresión de JSP (línea 26). Observe que en esta JSP aparecen tres estilos de comentarios (en las líneas 5, 17 y 23).

Los scriptlets definen una estructura `if...else` que determina si la JSP recibió un valor para el nombre de pila como parte de la petición. En la línea 19 se utiliza el método `getParameter` del objeto implícito de JSP `request` (un objeto `HttpServletRequest`) para obtener el valor del parámetro `nombrePila`, y se asigna el resultado a la variable `nombre`. En la línea 21 se determina si `nombre` no es `null` (es decir, que se haya pasado un valor para el nombre de pila a la JSP, como parte de la petición). Si esta condición es `true`, el scriptlet termina temporalmente, de manera que puedan enviarse a la salida los datos de plantilla fija de las líneas 25 a 28. La expresión de JSP de la línea 26 envía a la salida el valor de la variable `nombre` (es decir, el nombre de pila que se pasa a la JSP como parámetro de petición). El scriptlet continúa en las líneas 30 a 35 con el corchete de cierre del cuerpo de la estructura `if` y con el comienzo de la parte de la estructura `if...else` que corresponde al `else`. Si la condición de la línea 21 es `false`, no se envían a la salida las líneas 25 a 28. En vez de ello, las líneas 37 a 43 envían a la salida un elemento `form`. El usuario puede escribir un nombre de pila en el elemento `form` y oprimir el botón `Enviar` para pedir la JSP de nuevo y que se ejecute el cuerpo de la estructura `if` (líneas 25 a 28).

Observación de ingeniería de software 25.5



Pueden intercalarse scriptlets, expresiones y datos de plantilla fija en una JSP para crear diferentes respuestas, con base en la información que se incluya en la petición de una JSP.

Tip para prevenir errores 25.1



Algunas veces es difícil depurar los errores en una JSP, ya que los números de línea reportados por un contenedor de JSP se refieren normalmente al servlet que representa la JSP traducida, no a los números de línea de la JSP original. Los entornos de desarrollo de programas como Sun One Studio 4 de Sun Microsystems, Inc. permiten que las JSPs se compilen en el entorno, de manera que usted pueda ver los mensajes de error de sintaxis. Estos mensajes incluyen la instrucción en el servlet que representa la JSP traducida, lo cual puede ser útil para determinar el error.

Tip para prevenir errores 25.2



Muchos contenedores de JSPs almacenan los servlets que representan a las JSPs traducidas. Por ejemplo, el directorio de instalación de Tomcat contiene un subdirectorio llamado `work`, en el cual usted podrá encontrar el código fuente de los servlets traducidos por Tomcat.

Para probar la figura 25.4 en Tomcat, copie el archivo `bienvenida.jsp` en el directorio `jsp` que creó en la sección 25.3. Abra su navegador Web y escriba el siguiente URL para probar la página `bienvenida.jsp`:

```
http://localhost:8080/cpej5/jsp/bienvenida.jsp
```

Cuando ejecute por primera vez la JSP, ésta mostrará el elemento `form` en el que usted podrá escribir su nombre de pila, ya que el URL anterior no le pasa un parámetro `nombrePila` a la JSP. Una vez que envíe su nombre de pila, su navegador deberá mostrar una pantalla similar a la segunda captura de pantalla de la figura 25.4. *Nota:* Al igual que con los servlets, es posible pasar argumentos de peticiones `get` como parte del URL. El siguiente URL proporciona el parámetro `nombrePila` a `bienvenida.jsp`:

```
http://localhost:8080/cpej5/jsp/bienvenida.jsp?nombrePila=Paul
```

25.6 Acciones estándar

Continuaremos nuestra discusión sobre las JSPs con las *acciones estándar de JSP* (figura 25.5). Estas acciones proporcionan a los que implementan JSPs el acceso a varias de las tareas más comunes que se realizan en una JSP, como el incluir contenido de otros recursos, reenviar peticiones hacia otros recursos e interactuar con JavaBeans. Los contenedores de JSPs procesan las acciones en tiempo de petición. Las acciones están delimitadas por las marcas `<jsp:`*acción*`>` y `</jsp:`*acción*`>`, en donde *acción* es el nombre de la acción estándar. En casos en los que no aparece nada entre las marcas de inicio y fin, puede usarse la sintaxis de elemento vacío

http://librosysolucionarios.net

Acción	Descripción
`<jsp:include>`	Incluye, en forma dinámica, otro recurso en una JSP. Al ejecutarse la JSP, el recurso al que se hace referencia se incluye y se procesa.
`<jsp:forward>`	Reenvía el procesamiento de la petición hacia otra JSP, servlet o página estática. Esta acción termina la ejecución de la JSP actual.
`<jsp:plugin>`	Permite agregar un componente complemento a una página, en forma de elemento `object` o `embed` de HTML específico para cada navegador. En el caso de un applet de Java, esta acción permite la descarga e instalación del *Complemento de Java (Java Plug-in)*, si no está ya instalado en el equipo cliente.
`<jsp:param>`	Se utiliza con las acciones `include`, `forward` y `plugin` para especificar pares nombre/valor adicionales de información para que sean utilizados por estas acciones.
Manipulación de JavaBeans	
`<jsp:useBean>`	Especifica que la JSP utiliza una instancia de JavaBean. Esta acción especifica el alcance del bean y le asigna un ID que pueden utilizar los componentes de secuencias de comandos para manipularlo.
`<jsp:setProperty>`	Establece una propiedad en la instancia de JavaBean especificada. Una característica especial de esta acción es que se relacionan automáticamente los parámetros de la petición con las propiedades del bean con el mismo nombre.
`<jsp:getProperty>`	Obtiene una propiedad en la instancia de JavaBean especificada y convierte el resultado en una cadena para enviarla en la respuesta.

Figura 25.5 Acciones estándar de JSP.

`<jsp:`*accion* `/>` de XML. En la figura 25.5 se sintetizan las acciones estándar de JSP. Utilizaremos las acciones en las siguientes subsecciones.

25.6.1 La acción `<jsp:include>`

Las JavaServer Pages soportan dos mecanismos de inclusión: la *acción* `<jsp:include>` y la *directiva* `include`. La acción `<jsp:include>` permite incluir contenido dinámico en una JavaServer Page, en tiempo de petición (no en tiempo de traducción, como es el caso con la directiva `include`). Si el recurso incluido cambia entre peticiones, la siguiente petición a la JSP que contenga la acción `<jsp:include>` incluirá el nuevo contenido del recurso. Por otro lado, la directiva `include` copia el contenido en la JSP una vez, en tiempo de traducción de JSP. Si el recurso incluido cambia, el nuevo contenido no se reflejará en la JSP que utilizó la directiva `include`, a menos que se vuelva a compilar esa JSP. En la figura 25.6 se describen los atributos de la acción `<jsp:include>`.

Observación de ingeniería de software 25.6



De acuerdo con la especificación JavaServer Pages 1.1, a un contenedor de JSPs se le permite determinar si ha cambiado un recurso incluido con la directiva `include`. De ser así, el contenedor puede recompilar la JSP que incluyó el recurso. Sin embargo, la especificación no proporciona un mecanismo para indicar un cambio en un recurso incluido al contenedor.

Atributo	Descripción
`page`	Especifica la ruta URI relativa del recurso a incluir. El recurso debe formar parte de la misma aplicación Web.
`flush`	Especifica si el búfer debe vaciarse después de realizar la inclusión. En JSP 1.1 se requiere que este atributo sea `true`.

Figura 25.6 Los atributos de la acción `<jsp:include>`.

Tip de rendimiento 25.2

La acción `<jsp:include>` es más flexible que la directiva `include`, pero requiere más sobrecarga cuando el contenido de la página cambia frecuentemente. Use la acción `<jsp:include>` solamente cuando sea necesario el contenido dinámico.

Error común de programación 25.3

Establecer el atributo `flush` de la acción `<jsp:include>` en `false` es un error en tiempo de traducción. Actualmente, el atributo `flush` soporta solamente valores `true`.

Error común de programación 25.4

No especificar el atributo `flush` de la acción `<jsp:include>` es un error en tiempo de traducción. Es obligatorio especificar este atributo.

Error común de programación 25.5

Especificar en una acción `<jsp:include>` una página que no forma parte de la misma aplicación Web es un error en tiempo de petición; la acción `<jsp:include>` no incluirá ningún contenido.

El siguiente ejemplo muestra la acción `<jsp:include>` utilizando cuatro recursos de XHTML y JSP que representan contenido estático y dinámico. La JavaServer Page `incluir.jsp` (figura 25.10) incluye otros tres recursos: `titular.html` (figura 25.7), `contenido.html` (figura 25.8) y `reloj2.jsp` (figura 25.9). La JavaServer Page `incluir.jsp` crea un documento de XHTML que contiene un elemento `table`, en el cual `titular.html` abarca dos columnas a lo largo de la parte superior del elemento `table`, `contenido.html` es la columna izquierda de la segunda fila y `reloj2.jsp` (una versión simplificada de la figura 25.1) es la columna derecha de la segunda fila. La figura 25.10 utiliza tres acciones `<jsp:include>` (líneas 38 a 39, 48 y 55 a 56) como el contenido en los elementos `td` del elemento `table`. Mediante el uso de dos documentos de XHTML y una JSP, en la figura 25.10 se muestra que las JSPs pueden incluir contenido tanto estático como dinámico. La ventana de salida en la figura 25.10 muestra el resultado de una petición de `incluir.jsp`.

La figura 25.9 (`reloj2.jsp`) muestra cómo determinar la configuración regional del cliente mediante un objeto `Locale` (paquete `java.util`) y utiliza ese objeto `Locale` para dar formato a un objeto `Date` mediante un objeto `DateFormat` (paquete `java.text`). En la línea 14 se invoca el método `getLocale` del objeto `request`, el cual devuelve la configuración regional del cliente mediante un objeto `Locale`. En las líneas 17 a 20 se invoca el método `static` `getDateTimeInstance` de `DateFormat` para obtener un objeto `DateFormat`. Los primeros dos argumentos indican que los formatos de fecha y hora deben ser `LONG` (las otras opciones son `FULL`, `MEDIUM`, `SHORT` y `DEFAULT`). El tercer argumento especifica la configuración

```
1  <!-- Fig. 25.7: titular.html                    -->
2  <!-- titular a incluir en otro documento -->
3  <div style = "width: 580px">
4     <p>
5        Seminarios en el sitio de trabajo sobre:<br />
6        Java(TM), C, C++, Visual Basic(R), Tecnología de objetos y <br />
7        Capacitación sobre programación en Internet y World Wide Web<br />
8        Impartidos en todo el mundo
9     </p>
10
11    <p>
12       <a href = "mailto:deitel@deitel.com">deitel@deitel.com</a>
13       <br />978.461.5880<br />12 Clock Tower Place, Suite 200,
14       Maynard, MA 01754
15    </p>
16 </div>
```

Figura 25.7 Titular (`titular.html`) a incluir a lo largo de la parte superior del documento de XHTML creado por la figura 25.10.

http://librosysolucionarios.net


```
1  <!-- Fig. 25.8: contenido.html                  -->
2  <!-- contenido a incluir en otro documento -->
3
4  <p><a href = "http://www.deitel.com/books/index.html">
5     Publicaciones/Librería
6  </a></p>
7
8  <p><a href = "http://www.deitel.com/whatsnew.html">
9     Novedades
10 </a></p>
11
12 <p><a href = "http://www.deitel.com/books/downloads.html">
13    Descargas/Recursos
14 </a></p>
15
16 <p><a href = "http://www.deitel.com/faq/index.html">
17    FAQ (Preguntas frecuentes)
18 </a></p>
19
20 <p><a href = "http://www.deitel.com/intro.html">
21    Quiénes somos
22 </a></p>
23
24 <p><a href = "http://www.deitel.com/index.html">
25    Inicio
26 </a></p>
27
28 <p>Envíe sus preguntas o comentarios sobre este sitio a
29    <a href = "mailto:deitel@deitel.com">
30       deitel@deitel.com
31    </a><br />
32    Copyright 1995-2003 por Deitel &amp;amp; Associates, Inc.
33    Todos los derechos reservados.
34 </p>
```

Figura 25.8 Índice de contenido (`contenido.html`) a incluir hacia abajo en el lado izquierdo del documento de XHTML creado por la figura 25.10.

```
1  <!-- Fig. 25.9: reloj2.jsp                      -->
2  <!-- fecha y hora a incluir en otro documento -->
3
4  <table>
5     <tr>
6        <td style = "background-color: black;">
7           <p class = "grande" style = "color: cyan; font-size: 3em;
8              font-weight: bold;">
9
10             <%-- script para determinar configuración regional del --%>
11             <%-- cliente y dar formato a la fecha de manera acorde --%>
12             <%
13                // obtener configuración regional del cliente
14                java.util.Locale confRegional = request.getLocale();
15
16                // obtener objeto DateFormat para la configuración regional del cliente
```

Figura 25.9 La JSP `reloj2.jsp` a incluir como contenido principal en el documento de XHTML creado por la figura 25.10. (Parte 1 de 2.)

```
17                java.text.DateFormat formatoFecha =
18                   java.text.DateFormat.getDateTimeInstance(
19                      java.text.DateFormat.LONG,
20                         java.text.DateFormat.LONG, confRegional );
21
22             %> <%-- fin de secuencia de comandos --%>
23
24             <%-- mostrar la fecha --%>
25             <%= formatoFecha.format( new java.util.Date() ) %>
26          </p>
27       </td>
28    </tr>
29 </table>
```

Figura 25.9 La JSP `reloj2.jsp` a incluir como contenido principal en el documento de XHTML creado por la figura 25.10. (Parte 2 de 2.)

regional (`Locale`) para la que el objeto `DateFormat` debe dar formato a la fecha. En la línea 25 se invoca el método `format` del objeto `DateFormat` para producir una representación `String` del objeto `Date`. El objeto `DateFormat` da formato a este objeto `String` según el objeto `Locale` especificado en las líneas 17 a 20. [*Nota:* Este ejemplo funciona para los lenguajes occidentales que utilizan el conjunto de caracteres ISO8859-1. Sin embargo, para los lenguajes que no utilizan este conjunto de caracteres, la JSP debe especificar el conjunto de caracteres apropiado utilizando la directiva de JSP `page` (sección 25.7.1). En el sitio `java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html`, Sun proporciona una lista de codificaciones de caracteres. El tipo de contenido de la respuesta define el conjunto de caracteres a utilizar en la misma. El tipo de contenido tiene la forma: *"tipoMime*`;charset=` *codificación"* (por ejemplo, `"text/html;charset=ISO-8859-1"`).]

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.10: incluir.jsp -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9     <head>
10       <title>Uso de jsp:include</title>
11
12       <style type = "text/css">
13          body {
14             font-family: tahoma, helvetica, arial, sans-serif;
15          }
16
17       table, tr, td {
18          font-size: .9em;
19          border: 3px groove;
20          padding: 5px;
21          background-color: #dddddd;
22       }
23    </style>
24    </head>
25
```

Figura 25.10 La JSP `incluir.jsp` incluye recursos mediante `<jsp:include>`. (Parte 1 de 2.)

http://librosysolucionarios.net


```
26 <body>
27    <table>
28       <tr>
29          <td style = "width: 160px; text-align: center">
30             <img src = "imagenes/logochico.png"
31                width = "140" height = "93"
32                alt = "Logotipo de Deitel & Associates" />
33          </td>
34
35          <td>
36
37             <%-- incluir titular.html en esta JSP --%>
38             <jsp:include page = "titular.html"
39                flush = "true" />
40
41          </td>
42       </tr>
43
44       <tr>
45          <td style = "width: 160px">
46
47             <%-- incluir contenido.html en esta JSP --%>
48             <jsp:include page = "contenido.html" flush = "true" />
49
50          </td>
51
52          <td style = "vertical-align: top">
53
54             <%-- incluir reloj2.jsp en esta JSP --%>
55             <jsp:include page = "reloj2.jsp"
56                flush = "true" />
57
58          </td>
59       </tr>
60    </table>
61 </body>
62 </html>
```

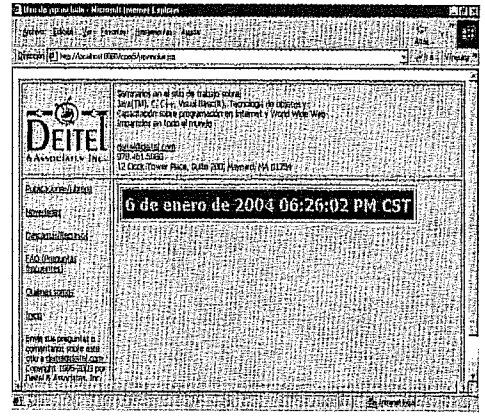


Figura 25.10 La JSP `incluir.jsp` incluye recursos mediante `<jsp:include>`. (Parte 2 de 2.)

Para probar la figura 25.10 en Tomcat, copie los archivos **titular.html**, **contenido.html**, **reloj2.jsp**, **incluir.jsp** y el directorio **imagenes** en el directorio **jsp** que creó en la sección 25.3. Abra su navegador Web y escriba el siguiente URL para probar la página **incluir.jsp**:

```
http://localhost:8080/cpej5/jsp/incluir.jsp
```

25.6.2 La acción <jsp:forward>

La acción ***<jsp:forward>*** permite a una JSP reenviar el procesamiento de la petición a un recurso distinto. El procesamiento de la petición por parte de la JSP original termina tan pronto como la JSP reenvía la petición. La acción **<jsp:forward>** sólo tiene un atributo **page** que especifica el URL relativo del recurso (en la misma aplicación Web) hacia el que deberá reenviarse la petición.



Observación de ingeniería de software 25.7

*Al utilizar la acción **<jsp:forward>**, el recurso al que se reenviará la petición debe encontrarse en el mismo contexto (aplicación Web) que la JSP que recibió originalmente la petición.*

La JavaServer Page **reenvio1.jsp** (figura 25.11) es una versión modificada de **bienvenida.jsp** (figura 25.4). La principal diferencia se encuentra en las línea 22 a 25, en las que la JavaServer Page **reenvio1.jsp** reenvía la petición hacia la JavaServer Page **reenvio2.jsp** (figura 25.12). Observe la acción **<jsp:param>** en las líneas 23 y 24. Esta acción agrega un parámetro de petición, el cual representa la fecha y hora en las que se recibió la petición original, al objeto petición que se reenvía a **reenvio2.jsp**.

La acción **<jsp:param>** especifica los pares nombre/valor de información que se pasan a las acciones **<jsp:include>**, **<jsp:forward>** y **<jsp:plugin>**. Cada acción **<jsp:param>** tiene dos atributos requeridos: **name** y **value**. Si una acción **<jsp:param>** especifica un parámetro que ya existe en la petición, el nuevo valor para ese parámetro tiene precedencia sobre el valor original. Todos los valores para ese parámetro pueden obtenerse mediante el uso del método **getParameterValues** del objeto implícito de JSP **request**, el cual devuelve un arreglo de objetos **String**.

La JSP **reenvio2.jsp** utiliza el atributo **name** especificado en la acción **<jsp:param>** (**"fecha"**) para obtener la fecha y hora. También utiliza el parámetro **nombrePila** que se pasó originalmente a **reenvio2.jsp** para obtener el nombre de pila del usuario. Las expresiones de JSP en la figura 25.12 (líneas 23 y 31) insertan los valores de los parámetros de petición en la respuesta al cliente. La captura de pantalla de la figura 25.11 muestra la interacción inicial con el cliente. La captura de pantalla de la figura 25.12 muestra los resultados devueltos al cliente después de reenviar la petición hacia **reenvio2.jsp**.

Para probar las figuras 25.11 y 25.12 en Tomcat, copie los archivos **reenvio1.jsp** y **reenvio2.jsp** en el directorio **jsp** que creó en la sección 25.3. Abra su navegador Web y escriba el siguiente URL para probar la página **reenvio1.jsp**:

```
http://localhost:8080/cpej5/jsp/reenvio1.jsp
```

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.11: reenvio1.jsp -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9  <head>
10    <title>Reenviar la petición hacia otra JSP</title>
11 </head>
```

Figura 25.11 La JSP **reenvio1.jsp** recibe un parámetro **nombrePila**, agrega una fecha a los parámetros de petición y reenvía la petición a **reenvio2.jsp** para continuar con su procesamiento. (Parte 1 de 2.)

http://librosysolucionarios.net


```
12
13 <body>
14    <% // iniciar scriptlet
15
16       String nombre = request.getParameter( "nombrePila" );
17
18       if ( nombre != null ) {
19
20    %> <%-- terminar scriptlet para insertar datos de plantilla fija --%>
21
22          <jsp:forward page = "reenvio2.jsp">
23             <jsp:param name = "fecha"
24                value = "<%= new java.util.Date() %>" />
25          </jsp:forward>
26
27    <% // continuar scriptlet
28
29       } // fin de instrucción if
30       else {
31
32    %> <%-- terminar scriptlet para insertar datos de plantilla fija --%>
33
34          <form action = "reenvio1.jsp" method = "get">
35             <p>Escriba su nombre de pila y oprima Enviar</p>
36
37             <p><input type = "text" name = "nombrePila" />
38                <input type = "submit" value = "Enviar" />
39             </p>
40          </form>
41
42    <% // continuar scriptlet
43
44       } // fin de instrucción else
45
46    %> <%-- fin de scriptlet --%>
47 </body>
48
49 </html> <!-- fin del documento de XHTML -->
```

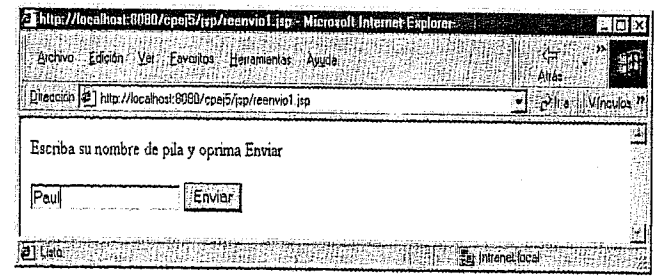


Figura 25.11 La JSP **reenvio1.jsp** recibe un parámetro **nombrePila**, agrega una fecha a los parámetros de petición y reenvía la petición a **reenvio2.jsp** para continuar con su procesamiento. (Parte 2 de 2.)

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- reenvio2.jsp -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml"v
8
9  <head>
10    <title>Procesamiento de una petición reenviada</title>
11
12    <style type = "text/css">
13       .grande {
14          font-family: tahoma, helvetica, arial, sans-serif;
15          font-weight: bold;
16          font-size: 2em;
17       }
18    </style>
19 </head>
20
21 <body>
22    <p class = "grande">
23       Hola <%= request.getParameter( "nombrePila" ) %>, <br />
24       Su petición fue recibida <br /> y reenviada el día y hora:
25    </p>
26
27    <table style = "border: 6px outset;">
28       <tr>
29          <td style = "background-color: black;">
30             <p class = "grande" style = "color: cyan;">
31                <%= request.getParameter( "fecha" ) %>
32             </p>
33          </td>
34       </tr>
35    </table>
36 </body>
37
38 </html>
```

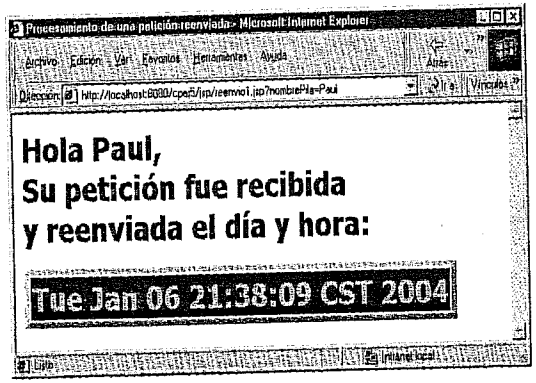


Figura 25.12 La JSP `reenvio2.jsp` recibe una petición (de `reenvio1.jsp` en este ejemplo) y utiliza los parámetros de respuesta como parte de la respuesta al cliente.

http://librosysolucionarios.net

25.6.3 La acción `<jsp:useBean>`

La acción `<jsp:useBean>` permite a una JSP manipular un objeto de Java. Esta acción crea un objeto de Java o localiza uno existente para utilizarlo en la JSP. En la figura 25.13 se sintetizan los atributos de la acción `<jsp:useBean>`. Si no se especifican los atributos `class` y `beanName`, el contenedor de JSPs trata de localizar un objeto existente del tipo especificado en el atributo `type`. Al igual que los objetos implícitos de JSP, los objetos que se especifican con la acción `<jsp:useBean>` tienen alcance tipo `page`, `request`, `session` o `application`, el cual indica en qué parte de una aplicación Web pueden utilizarse. Los objetos con alcance tipo `page` pueden ser utilizados solamente por la página en la que están definidos. Varias páginas de JSP pueden potencialmente acceder a los objetos en otros alcances. Por ejemplo, todas las JSPs que procesan una sola petición pueden acceder a un objeto con alcance tipo `request`.



Error común de programación 25.6

Debe especificarse uno de los atributos `class` o `type` de `<jsp:useBean>`, o ambos; de no ser así, se producirá un error en tiempo de traducción.

Muchos sitios Web colocan anuncios rotativos en sus páginas Web. Cada visita a una de estas páginas normalmente hace que se muestre un anuncio distinto en el navegador Web del usuario. Por lo general, si hace clic en un anuncio será llevado al sitio Web de la compañía que colocó ese anuncio. Nuestro primer ejemplo de `<jsp:useBean>` muestra un bean simple para rotar anuncios, que itera a través de una lista de cinco anuncios. En este ejemplo, los anuncios son portadas de algunos de nuestros libros. Al hacer un clic en una portada será llevado al sitio Web de Amazon, en donde podrá leer acerca del libro y posiblemente ordenarlo.

El bean `Rotador` (figura 25.14) tiene tres métodos: `getImagen`, `getVinculo` y `siguienteAnuncio`. El método `getImagen` devuelve el nombre del archivo con la imagen de la cubierta de un libro. El método `getVinculo` devuelve el hipervínculo que conduce al libro en Amazon.com. El método `siguienteAnuncio` actualiza el `Rotador`, de manera que las siguientes llamadas a `getImagen` y `getVinculo` devuelvan la información de un anuncio distinto. Cada uno de los métodos `getImagen` y `getVinculo` representa una propiedad de JavaBean de sólo lectura: `imagen` y `vinculo`, respectivamente. `Rotador` lleva el registro del anuncio actual con su variable `indiceSeleccionado`, la cual se actualiza mediante una invocación al método `siguienteAnuncio`.

En las líneas 7 y 8 de la JavaServer Page `rotadoranuncios.jsp` (figura 25.15) se obtiene una referencia a una instancia de la clase `Rotador`. El identificador para el bean es `rotador`. La JSP utiliza este nombre para manipular el bean. El alcance del objeto es `session`, por lo que cada cliente individual verá la misma secuencia de anuncios durante su sesión de navegación. Cuando `rotadoranuncios.jsp` reciba una petición de un nuevo cliente, el contenedor de JSPs crea el bean y lo almacena en el objeto `session` de ese cliente (un objeto `HttpSession`). En cada petición a esta JSP, la línea 22 utiliza la referencia `rotador` crea-

Atributo	Descripción
`id`	El nombre utilizado para manipular el objeto de Java con las acciones `<jsp:setProperty>` y `<jsp:getProperty>`. Una variable de este nombre también se declara para usarse en los elementos de secuencias de comandos de JSP. El nombre que se especifica aquí es susceptible al uso de mayúsculas y minúsculas.
`scope`	El alcance en el cual el objeto de Java puede utilizarse: `page`, `request`, `session` o `application`. El alcance predeterminado es `page`.
`class`	El nombre completamente calificado de la clase del objeto de Java.
`beanName`	El nombre de un bean que puede utilizarse con el método `instantiate` de la clase `java.beans.Beans` para cargar un JavaBean en memoria.
`type`	El tipo del JavaBean. Éste puede ser el mismo tipo que el atributo `class`, una superclase de ese tipo o una interfaz implementada por ese tipo. El valor predeterminado es el mismo que para el atributo `class`. Se produce una excepción `ClassCastException` si el objeto de Java no es del tipo especificado con el atributo `type`.

Figura 25.13 Atributos de la acción `<jsp:useBean>`.

```
1  // Fig. 25.14: Rotador.java
2  // Un JavaBean que rota anuncios.
3  package com.deitel.cpej5.jsp;
4
5  public class Rotador {
6     private String imagenes[] = { "imagenes/advjHTP1.jpg",
7        "imagenes/cppHTP4.jpg", "imagenes/iw3HTP2.jpg",
8        "imagenes/jwsFEP1.jpg", "imagenes/vbnetHTP2.jpg" };
9
10    private String vinculos[] = {
11       "http://www.amazon.com/exec/obidos/ASIN/0130895601/" +
12          "deitelassociatin",
13       "http://www.amazon.com/exec/obidos/ASIN/0130384747/" +
14          "deitelassociatin",
15       "http://www.amazon.com/exec/obidos/ASIN/0130308978/" +
16          "deitelassociatin",
17       "http://www.amazon.com/exec/obidos/ASIN/0130461342/" +
18          "deitelassociatin",
19       "http://www.amazon.com/exec/obidos/ASIN/0130293636/" +
20          "deitelassociatin" };
21
22    private int indiceSeleccionado = 0;
23
24    // devuelve el nombre del archivo de imagen para el anuncio actual
25    public String getImagen()
26    {
27       return imagenes[ indiceSeleccionado ];
28    }
29
30    // devuelve el URL para el sitio Web correspondiente al anuncio
31    public String getVinculo()
32    {
33       return vinculos[ indiceSeleccionado ];
34    }
35
36    // actualizar indiceSeleccionado para que las siguientes llamadas a getImagen
37    // y getVinculo devuelvan un anuncio distinto
38    public void siguienteAnuncio()
39    {
40       indiceSeleccionado = ( indiceSeleccionado + 1 ) % imagenes.length;
41    }
42 }
```

Figura 25.14 El bean `Rotador` para manipular un conjunto de anuncios.

da en la línea 7 para invocar el método `siguienteAnuncio` del bean `Rotador`. Por lo tanto, cada petición recibirá el siguiente anuncio seleccionado por el bean `Rotador`. En las líneas 29 a 34 se define un hipervínculo al sitio Amazon.com para cada libro. En las líneas 29 y 30 se introduce la acción `<jsp:getProperty>` para obtener el valor de la propiedad `vinculo` del bean `Rotador`. La acción `<jsp:getProperty>` tiene dos atributos (`name` y `property`) que especifican el objeto bean que se va a manipular y la propiedad que se va a obtener. Si el objeto JavaBean utiliza las convenciones de nomenclatura estándar de JavaBean, el método usado para obtener el valor de la propiedad `vinculo` del bean deberá ser `getVinculo`. La acción `<jsp:getProperty>` invoca a `getVinculo` en el bean al que se hace referencia con `rotador`, convierte el valor de retorno en un objeto `String` y envía ese objeto como parte de la respuesta al cliente. La propiedad `vinculo` se convierte en el valor del atributo `href` del hipervínculo. Este hipervínculo se representa en la página Web

resultante como la imagen de la cubierta del libro. En las líneas 32 y 33 se crea un elemento `img` y se utiliza otra acción `<jsp:getProperty>` para obtener el valor de la propiedad `imagen` del bean `Rotador`.

Las propiedades `vinculo` e `imagen` pueden obtenerse también mediante expresiones de JSP. Por ejemplo, la acción `<jsp:getProperty>` en las líneas 29 y 30 puede reemplazarse con la expresión:

```
<%= rotador.getVinculo() %>
```

De manera similar, la acción `<jsp:getProperty>` en las líneas 32 y 33 puede reemplazarse con la expresión:

```
<%= rotador.getImagen() %>
```

Para probar `rotadoranuncios.jsp` en Tomcat, copie el archivo `rotadoranuncios.jsp` en el directorio `jsp` que creó en la sección 25.3. Debió haber copiado el directorio `imagenes` en el directorio `jsp` cuando probó la figura 25.10. Si no lo hizo, debe copiar ahora el directorio `imagenes` ahí. Copie el archivo `Rotador.class` en el directorio `WEB-INF\classes` de la aplicación Web `cpej5` en Tomcat. [*Nota:* Este ejemplo funcionará solamente si se define la estructura de directorios del paquete apropiada para `Rotador`,

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.15: rotadoranuncios.jsp -->
6
7  <jsp:useBean id = "rotador" scope = "session"
8     class = "com.deitel.cpej5.jsp.Rotador" />
9
10 <html xmlns = "http://www.w3.org/1999/xhtml">
11
12    <head>
13       <title>Ejemplo de rotador de anuncios</title>
14
15       <style type = "text/css">
16          .grande { font-family: helvetica, arial, sans-serif;
17                    font-weight: bold;
18                    font-size: 2em }
19       </style>
20
21       <%-- actualizar anuncio --%>
22       <% rotador.siguienteAnuncio(); %>
23    </head>
24
25    <body>
26       <p class = "grande">Ejemplo de rotador de anuncios</p>
27
28       <p>
29          <a href = "<jsp:getProperty name = "rotador"
30             property = "vinculo" />">
31
32             <img src = "<jsp:getProperty name = "rotador"
33               property = "imagen" />" alt = "anuncio" />
34          </a>
35       </p>
36    </body>
37 </html>
```

Figura 25.15 La JSP `rotadoranuncios.jsp` utiliza un bean `Rotador` para mostrar un anuncio distinto en cada petición de la página. (Parte 1 de 2.)

http://librosysolucionarios.net

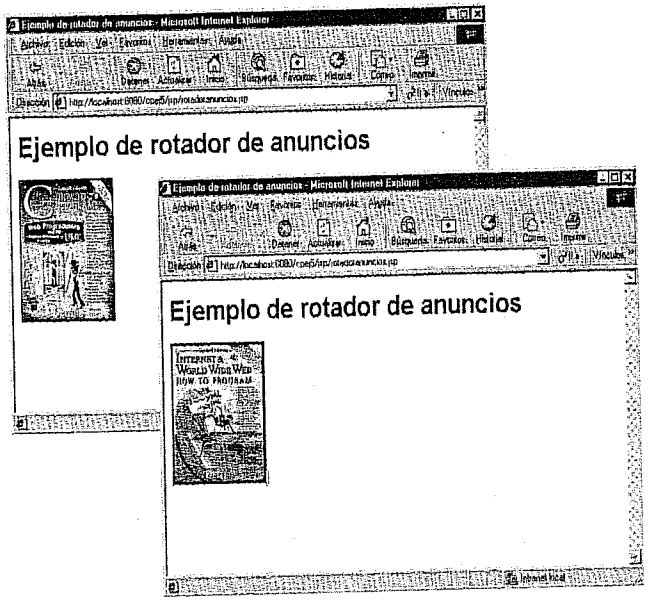


Figura 25.15 La JSP `rotadoranuncios.jsp` utiliza un bean `Rotador` para mostrar un anuncio distinto en cada petición de la página. (Parte 2 de 2.)

en el directorio `classes`. `Rotador` se define en el paquete `com.deitel.cpej5.jsp`. Abra su navegador Web y escriba el siguiente URL para probar la página `rotadoranuncios.jsp`:

```
http://localhost:8080/cpej5/jsp/rotadoranuncios.jsp
```

Pruebe a volver a cargar esta JSP varias veces en su navegador, para que vea el cambio de anuncios con cada petición.

La acción `<jsp:setProperty>` establece los valores de las propiedades de un JavaBean, y es particularmente útil para asociar los valores de los parámetros de una petición con las propiedades de un JavaBean. Los parámetros de petición pueden utilizarse para establecer propiedades que tengan los tipos primitivos `boolean`, `byte`, `char`, `int`, `long`, `float` y `double`, y los tipos `String`, `Boolean`, `Byte`, `Character`, `Integer`, `Long`, `Float` y `Double` de `java.lang`. En la figura 25.16 se sintetizan los atributos de la acción `<jsp:setProperty>`.



Error común de programación 25.7

Utilice el atributo `value` de la acción `<jsp:setProperty>` para establecer tipos de propiedades de un JavaBean que no puedan establecerse con parámetros de petición; en caso contrario se producirán errores de conversión.



Observación de ingeniería de software 25.8

La acción `<jsp:setProperty>` puede utilizar valores de parámetros de petición para establecer propiedades de un JavaBean, solamente en propiedades con los siguientes tipos: objetos `String`, tipos primitivos (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`) y clases de envoltura de tipos (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` y `Double`).

25.7 Directivas

Las *directivas* son mensajes para el contenedor de JSPs, que permiten al programador especificar información acerca de la página (como la página de error), incluir contenido de otros recursos y especificar bibliotecas de marcas personalizadas para usarlas en una JSP. Las directivas (delimitadas por `<%@` y `%>`) se procesan en tiempo de traducción. Por lo tanto, las directivas no producen ningún tipo de salida inmediata, ya que se procesan antes de que la JSP acepte cualquier petición. En la figura 25.17 se sintetizan los tres tipos de directivas. En las siguientes subsecciones hablaremos sobre estas directivas.

Atributo	Descripción
`name`	El identificador (ID) del JavaBean para el cual se va a establecer una propiedad (o propiedades).
`property`	El nombre de la propiedad a establecer. Si se especifica `"*"` para este atributo, la JSP relacionará los parámetros de la petición con las propiedades del bean. Por cada parámetro de petición que concuerde (es decir, que el nombre del parámetro de la petición sea igual al nombre de la propiedad del bean) se asignará a la propiedad correspondiente en el bean el valor del parámetro. Si el valor del parámetro de petición es `""`, el valor de la propiedad en el bean permanecerá sin cambio.
`param`	Si los nombres de los parámetros de la petición no concuerdan con los nombres de las propiedades del bean, este atributo puede utilizarse para especificar cuál parámetro de petición debe usarse para obtener el valor de una propiedad de bean específica. Este atributo es opcional. Si se omite, los nombres de los parámetros de la petición deberán concordar con los nombres de las propiedades del bean.
`value`	El valor que se va a asignar a una de las propiedades del bean. Este valor generalmente es el resultado de una expresión de JSP. Este atributo es especialmente útil para establecer las propiedades del bean que no pueden establecerse mediante los parámetros de la petición. Este atributo es opcional. Si se omite, la propiedad del JavaBean deberá ser de un tipo que pueda establecerse mediante los parámetros de petición.

Figura 25.16 Atributos de la acción `<jsp:setProperty>`.

25.7.1 La directiva page

La *directiva `page`* especifica las configuraciones globales de la JSP en el contenedor de JSPs. Puede haber muchas directivas `page`, siempre y cuando sólo haya una ocurrencia de cada atributo. La única excepción a esto es el atributo `import`, el cual puede usarse repetidamente para importar los paquetes de Java que se utilicen en la JSP. En la figura 25.18 se sintetizan los atributos de la directiva `page`.



Error común de programación 25.8

Proporcionar varias directivas `page` con uno o más atributos repetidos en común es un error de JSP en tiempo de traducción. Además, proporcionar una directiva `page` con un atributo o valor que no sea reconocido es un error de JSP en tiempo de traducción.

Directiva	Descripción
`page`	Define las configuraciones de página para que las procese el contenedor de JSP.
`include`	Hace que el contenedor de JSPs inserte en tiempo de traducción el contenido de otro recurso. A medida que la JSP se traduce en un servlet y se compila, el archivo al que se hace referencia sustituye la directiva y se traduce como si originalmente hubiera formado parte de la JSP.
`taglib`	Permite a los programadores definir nuevas marcas en forma de *bibliotecas de marcas*, las cuales pueden utilizarse para encapsular la funcionalidad y simplificar la codificación de una JSP.

Figura 25.17 Directivas de JSP.

http://librosysolucionarios.net

Atributo	Descripción
`language`	El lenguaje de secuencias de comandos utilizado en la JSP. Hasta este momento, el único valor permitido en este atributo es `java`.
`extends`	Especifica la clase a partir de la cual se heredará la JSP traducida. Este atributo debe ser un nombre de clase completamente calificado.
`import`	Especifica una lista separada por comas de los nombres completamente calificados de los tipos y/o paquetes que se utilizarán en la JSP actual. Cuando el lenguaje de secuencias de comandos es `java`, la lista predeterminada de `import` es `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*` y `javax.servlet.http.*`. Si se especifican varias propiedades `import`, el contenedor coloca en una lista los nombres de los paquetes.
`session`	Especifica si la página participa en una sesión. Los valores para este atributo son `true` (participa en una sesión; es el valor predeterminado) o `false` (no participa en una sesión). Cuando la página forma parte de una sesión, el objeto implícito `session` está disponible para ser utilizado en esa página. En caso contrario, `session` no está disponible y, si se utiliza en el código de la secuencia de comandos, se genera un error en tiempo de traducción.
`buffer`	Especifica el tamaño del búfer de salida utilizado con el objeto implícito `out`. El valor de este atributo puede ser `none` cuando no se usa búfer, o puede ser un valor como `8kb` (el tamaño de búfer predeterminado). La especificación JSP indica que el búfer utilizado debe ser cuando menos del tamaño especificado.
`autoFlush`	Cuando se establece en `true` (el valor predeterminado), este atributo indica que el búfer de salida utilizado con el objeto implícito `out` debe vaciarse automáticamente al llenarse el búfer. Si se establece en `false`, se produce una excepción si el búfer se desborda. El valor de este atributo debe ser `true` si el atributo `buffer` se establece en `none`.
`isThreadSafe`	Especifica si la página es segura para usar subprocesos. Si es `true` (el valor predeterminado), la página se considera como segura para usar subprocesos, y puede procesar varias peticiones al mismo tiempo. Si es `false`, el servlet que representa a la página implementa a la interfaz `java.lang.SingleThreadModel` y esa JSP sólo puede procesar una petición a la vez. El estándar JSP permite que existan varias instancias de una JSP, en el caso de las JSPs que no son seguras para usar subprocesos. Esto permite al contenedor manejar las peticiones de una manera más eficiente. Sin embargo, esto no garantiza que los recursos compartidos entre las instancias de la JSP se accedan en una manera segura para usar subprocesos.
`info`	Especifica una cadena de información que describe la página. Esta cadena es devuelta por el método `getServletInfo` del servlet que representa a la JSP traducida. Este método puede invocarse a través del objeto implícito `page` de la JSP.
`errorPage`	Cualquier excepción en la página actual que no se atrape, se envía a la página de error para su procesamiento. El objeto implícito `exception` de la página de error hace referencia a la excepción original.
`isErrorPage`	Especifica si la página actual es una página de error que será invocada en respuesta a un error en otra página. Si el valor del atributo es `true`, se crea el objeto implícito `exception` y hace referencia a la excepción original que ocurrió. Si es `false` (la opción predeterminada), cualquier uso del objeto `exception` en la página producirá un error en tiempo de traducción.
`contentType`	Especifica el tipo MIME de los datos en la respuesta al cliente. El tipo predeterminado es `text/html`.

Figura 25.18 Atributos de la directiva `page`.



Observación de ingeniería de software 25.9

De acuerdo con la sección 2.7.1 de la especificación JSP, el atributo `extends` "no debe utilizarse sin considerarlo cuidadosamente, ya que restringe la habilidad del contenedor de JSPs de proporcionar superclases especializadas que pueden mejorar la calidad del servicio desempeñado". Recuerde que una clase de Java puede extender solamente a una clase. Si su JSP especifica una superclase explícita, el contenedor de JSP no podrá incluir su JSP en una subclase de una de las propias clases de servlet mejoradas de la aplicación contenedora.



Error común de programación 25.9

El uso del objeto implícito de JSP `session` en una JSP que no tenga su atributo `session` de la directiva `page` establecido en `true` es un error en tiempo de traducción.

25.7.2 La directiva `include`

La *directiva* `include` incluye el contenido de otro recurso una vez, en tiempo de traducción de la JSP. La directiva `include` sólo tiene un atributo (`file`), el cual especifica el URL del recurso a incluir. La diferencia entre la directiva `include` y la acción `<jsp:include>` se nota solamente si cambia el contenido incluido. Por ejemplo, si la definición de un documento de XHTML cambia después de ser incluido mediante la directiva `include`, las futuras invocaciones de la JSP mostrarán el contenido original del documento de XHTML, no el nuevo documento. En contraste, la acción `<jsp:include>` se procesa en cada petición de la JSP. Por lo tanto, los cambios al contenido incluido serían aparentes en la siguiente petición de la JSP que utilice la acción `<jsp:include>`.

La JSP `directivaInclude.jsp` (figura 25.19) es una reimplementación de `incluir.jsp` (figura 25.10), utilizando directivas `include`. Para probar `directivaInclude.jsp` en Tomcat, copie el archivo `directivaInclude.jsp` en el directorio `jsp` que creó en la sección 25.3. Abra su navegador Web y escriba el siguiente URL para probar la página `directivaInclude.jsp`:

```
http://localhost:8080/cpej5/jsp/directivaInclude.jsp
```

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.19: directivaInclude.jsp -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9     <head>
10       <title>Uso de la directiva include</title>
11
12       <style type = "text/css">
13          body {
14             font-family: tahoma, helvetica, arial, sans-serif;
15          }
16
17          table, tr, td {
18             font-size: .9em;
19             border: 3px groove;
20             padding: 5px;
21             background-color: #dddddd;
22          }
23       </style>
24    </head>
25
26    <body>
27       <table>
28          <tr>
29             <td style = "width: 160px; text-align: center">
30                <img src = "imagenes/logochico.png"
31                   width = "140" height = "93"
32                   alt = "Logotipo de Deitel & Associates, Inc." />
```

Figura 25.19 La JSP `directivaInclude.jsp` muestra cómo incluir contenido en tiempo de traducción mediante la directiva `include`. (Parte 1 de 2.)

http://librosysolucionarios.net

```
33            </td>
34
35            <td>
36
37               <%-- incluir titular.html en esta JSP --%>
38               <%@ include file = "titular.html" %>
39
40            </td>
41         </tr>
42
43      <tr>
44            <td style = "width: 160px">
45
46               <%-- incluir contenido.html en esta JSP --%>
47               <%@ include file = "contenido.html" %>
48
49            </td>
50
51            <td style = "vertical-align: top">
52
53               <%-- incluir reloj2.jsp en esta JSP --%>
54               <%@ include file = "reloj2.jsp" %>
55
56            </td>
57         </tr>
58      </table>
59   </body>
60 </html>
```

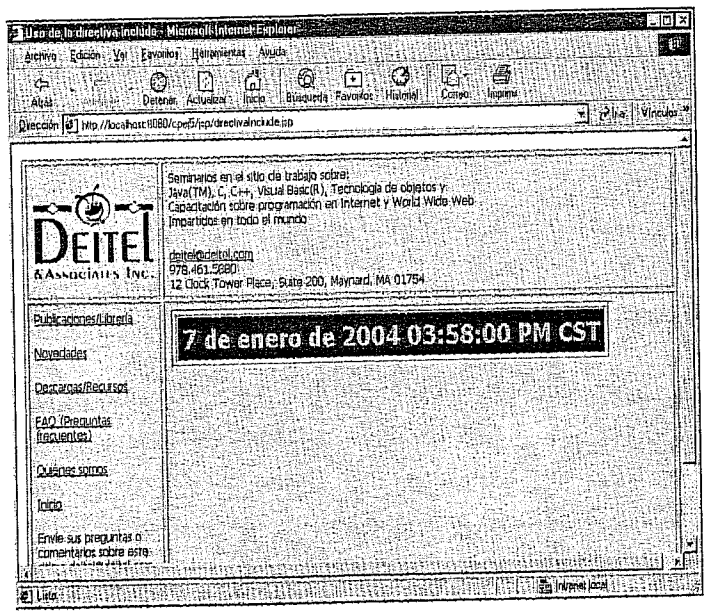


Figura 25.19 La JSP **directivaInclude.jsp** muestra cómo incluir contenido en tiempo de traducción mediante la directiva **include**. (Parte 2 de 2.)

http://librosysolucionarios.net

25.8 Ejemplo práctico: Libro de visitantes

Nuestro siguiente ejemplo es un libro de visitantes que permite a los usuarios colocar su nombre de pila, apellido paterno y dirección de e-mail en una base de datos. Después de enviar su información, los usuarios ven una página Web que contiene a todos los usuarios en el libro. La dirección de e-mail de cada persona se muestra como un hipervínculo que permite al usuario enviar un mensaje de correo electrónico a esa persona. Este ejemplo muestra el uso de la acción **<jsp:setProperty>**, la directiva **page** de JSP, las páginas de error de JSP y el uso de JDBC.

El ejemplo del libro de visitantes consiste en los JavaBeans **BeanVisitante** (figura 25.20) y **BeanDatosVisitante** (figura 25.21), y en las JSPs **inicioLibroVisitantes.jsp** (figura 25.22), **vistaLibroVisitantes.jsp** (figura 25.23) y **paginaErrorLibroVisitantes.jsp** (figura 25.24). En la figura 25.25 se muestran ventanas con los resultados de este ejemplo. El JavaBean **BeanVisitante** (figura 25.20) define tres propiedades de los visitantes: **nombrePila**, **apellidoPaterno** y **email**. Cada una es una propiedad de lectura/escritura que cuenta con métodos *set* y *get* para manipularla.

```
1  // Fig. 25.20: BeanVisitante.java
2  // JavaBean para almacenar los datos de un visitante en el libro de visitantes.
3  package com.deitel.cpej5.jsp.beans;
4
5  public class BeanVisitante {
6     private String nombrePila, apellidoPaterno, email;
7
8     // establecer el nombre de pila del visitante
9     public void setNombrePila( String nombre )
10    {
11       nombrePila = nombre;
12    }
13
14    // obtener el nombre de pila del visitante
15    public String getNombrePila()
16    {
17       return nombrePila;
18    }
19
20    // establecer el apellido paterno del visitante
21    public void setApellidoPaterno( String nombre )
22    {
23       apellidoPaterno = nombre;
24    }
25
26    // obtener el apellido paterno del visitante
27    public String getApellidoPaterno()
28    {
29       return apellidoPaterno;
30    }
31
32    // establecer la dirección de email del visitante
33    public void setEmail( String direccion )
34    {
35       email = direccion;
36    }
37
38    // obtener la dirección de email del visitante
```

Figura 25.20 **BeanVisitante** almacena la información de un visitante. (Parte 1 de 2.)

```
39     public String getEmail()
40     {
41        return email;
42     }
43  }
```

Figura 25.20 **BeanVisitante** almacena la información de un visitante. (Parte 2 de 2.)

El JavaBean **BeanDatosVisitante** (figura 25.21) se conecta con la base de datos **librovisitantes** y proporciona los métodos **getListaVisitantes** y **agregarVisitante** para manipular la base de datos. La base de datos del libro de visitantes tiene una sola tabla (**visitantes**), la cual contiene tres columnas (**nombrePila**, **apellidoPaterno** y **email**). Proporcionamos una secuencia de comandos de SQL (**librovisitantes.sql**) con este ejemplo, la cual puede utilizarse con el DBMS Cloudscape para crear la base de datos **librovisitantes**. Para obtener más detalles acerca de cómo crear una base de datos con Cloudscape, consulte el capítulo 23.

```
1  // Fig. 25.21: BeanDatosVisitante.java
2  // La clase BeanDatosVisitante realiza una conexión a una base de datos y soporta
3  // las operaciones de insertar y recuperar información de esa base de datos.
4  package com.deitel.cpej5.jsp.beans;
5
6  // paquetes básicos de Java
7  import java.io.*;
8  import java.sql.*;
9  import java.util.*;
10
11 public class BeanDatosVisitante {
12    private Connection conexion;
13    private Statement instruccion;
14
15    // construir objeto BeanDatosVisitante
16    public BeanDatosVisitante() throws Exception
17    {
18          System.setProperty( "db2j.system.home", "C:/Cloudscape_5.1" );
19
20          // cargar el controlador de Cloudscape
21          Class.forName("com.ibm.db2j.jdbc.DB2jDriver");
22
23          // conectarse a la base de datos
24          conexion = DriverManager.getConnection(
25             "jdbc:db2j:librovisitantes" );
26
27          instruccion = conexion.createStatement();
28    }
29
30    // devolver un objeto ArrayList de objetos BeanVisitante
31    public List getListaVisitantes() throws SQLException
32    {
33       List listaVisitantes = new ArrayList();
34
35       // obtener la lista de los títulos
```

Figura 25.21 **BeanDatosVisitante** accede a la base de datos a beneficio de **inicioLibroDeVisitantes.jsp**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
36       ResultSet resultados = instruccion.executeQuery(
37          "SELECT nombrePila, apellidoPaterno, email FROM visitantes" );
38
39       // obtener datos de la fila
40       while ( resultados.next() ) {
41       BeanVisitante visitante = new BeanVisitante();
42
43          visitante.setNombrePila( resultados.getString( 1 ) );
44          visitante.setApellidoPaterno( resultados.getString( 2 ) );
45          visitante.setEmail( resultados.getString( 3 ) );
46
47          listaVisitantes.add( visitante );
48       }
49
50       return listaVisitantes;
51    }
52
53    // insertar un visitante en la base de datos del libro de visitantes
54    public void agregarVisitante( BeanVisitante visitante ) throws SQLException
55    {
56       instruccion.executeUpdate( "INSERT INTO visitantes ( nombrePila, " +
57          "apellidoPaterno, email ) VALUES ( '" + visitante.getNombrePila() + "', '" +
58          visitante.getApellidoPaterno() + "', '" + visitante.getEmail() + "' )" );
59    }
60
61    // cerrar instrucciones y terminar la conexión a la base de datos
62    protected void finalize()
63    {
64       // tratar de cerrar la conexión a la base de datos
65       try {
66          instruccion.close();
67          conexion.close();
68       }
69
70       // procesar posible excepción SQLException en operación de cierre
71       catch ( SQLException excepcionSQL ) {
72        excepcionSQL.printStackTrace();
73       }
74    }
75 }
```

Figura 25.21 **BeanDatosVisitante** accede a la base de datos a beneficio de **inicioLibroDeVisitantes.jsp**. (Parte 2 de 2.)

El método **getListaVisitantes** de **BeanDatosVisitante** (líneas 31 a 51) devuelve un objeto **ArrayList** de objetos **BeanVisitante**, que representan a los invitados en la base de datos. El método **getListaVisitantes** crea los objetos **BeanVisitante** del objeto **ResultSet** devuelto por el método **executeQuery** de **Statement** (líneas 36 y 37).

El método **agregarVisitante** de **BeanDatosVisitante** (líneas 54 a 59) recibe un objeto **BeanVisitante** como argumento y utiliza las propiedades del **BeanInvitado** como argumentos para el método **executeUpdate** de **Statement** (líneas 56 a 58). Este objeto **Statement** inserta un nuevo invitado en la base de datos.

Observe que ni el constructor de **BeanDatosVisitante**, ni los métodos **getListaVisitantes** y **agregarVisitante** procesan las posibles excepciones. En el constructor, la línea 21 puede lanzar una excepción **ClassNotFoundException** y las otras instrucciones pueden lanzar excepciones **SQLExcep-**

tion. De manera similar, pueden lanzarse excepciones **SQLException** desde los cuerpos de los métodos **getListaVisitantes** y **agregarVisitante**. En este ejemplo permitimos intencionalmente que cualquier excepción que se produzca pase de vuelta a la JSP que invoca al constructor o a los métodos de **BeanDatosVisitante**. Esto nos permite mostrar el uso de las páginas de error de JSP. Cuando una JSP realiza una operación que produce una excepción, puede incluir scriptlets para atrapar la excepción y procesarla. Las excepciones que no se atrapen pueden reenviarse a una página de error de JSP para que ésta las maneje.

La JavaServerPage **inicioLibroVisitantes.jsp** (figura 25.22) es una versión modificada de **reenvio1.jsp** (figura 25.11), que muestra un elemento **form** en donde los usuarios pueden escribir su nombre de pila, apellido paterno y dirección de e-mail. Cuando el usuario envía el elemento **form** se vuelve a pedir la JSP **inicioLibroVisitantes.jsp**, de manera que pueda utilizar todos los valores de los datos introducidos. De no ser así, **inicioLibroVisitantes.jsp** responde con el elemento **form** otra vez, para que el usuario pueda llenar el (los) campo(s) faltante(s). Si el usuario proporciona las tres piezas de información requeridas, **inicioLibroVisitantes.jsp** reenvía la petición hacia **vistaLibroVisitantes.jsp**, la cual muestra el contenido del libro de visitantes.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.22: inicioLibroVisitantes.jsp -->
6
7  <%-- configuración de página --%>
8  <%@ page errorPage = "paginaErrorLibroVisitantes.jsp" %>
9
10 <%-- beans utilizados en esta JSP --%>
11 <jsp:useBean id = "visitante" scope = "page"
12    class = "com.deitel.cpej5.jsp.beans.BeanVisitante" />
13 <jsp:useBean id = "datosVisitante" scope = "request"
14    class = "com.deitel.cpej5.jsp.beans.BeanDatosVisitante" />
15
16 <html xmlns = "http://www.w3.org/1999/xhtml">
17
18 <head>
19    <title>Inicio de sesión del libro de visitantes</title>
20
21    <style type = "text/css">
22       body {
23          font-family: tahoma, helvetica, arial, sans-serif;
24       }
25
26       table, tr, td {
27          font-size: .9em;
28          border: 3px groove;
29          padding: 5px;
30          background-color: #dddddd;
31       }
32    </style>
33 </head>
34
35 <body>
36    <jsp:setProperty name = "visitante" property = "*" />
```

Figura 25.22 La JavaServer Page **inicioLibroVisitantes.jsp** permite al usuario enviar un nombre de pila, apellido paterno y dirección de e-mail para colocarlas en el libro de visitantes. (Parte 1 de 3.)


```
37
38    <% // iniciar scriptlet
39
40       if ( visitante.getNombrePila() == null ||
41            visitante.getApellidoPaterno() == null ||
42            visitante.getEmail() == null ) {
43
44    %> <%-- terminar scriptlet para insertar datos de plantilla fija --%>
45
46          <form method = "post" action = "inicioLibroVisitantes.jsp">
47             <p>Escriba su nombre de pila, apellido paterno y dirección
48              de e-mail para registrarlo en nuestro libro de visitantes.</p>
49
50             <table>
51                <tr>
52                   <td>Nombre de pila</td>
53
54                   <td>
55                      <input type = "text" name = "nombrePila" />
56                   </td>
57                </tr>
58
59                <tr>
60                   <td>Apellido paterno</td>
61
62                   <td>
63                      <input type = "text" name = "apellidoPaterno" />
64                   </td>
65                </tr>
66
67                <tr>
68                   <td>E-mail</td>
69
70                   <td>
71                      <input type = "text" name = "email" />
72                   </td>
73                </tr>
74
75                <tr>
76                   <td colspan = "2">
77                      <input type = "submit"
78                         value = "Enviar" />
79                   </td>
80                </tr>
81             </table>
82          </form>
83
84    <% // continuar scriptlet
85
86       } // fin de instrucción if
87       else {
88          datosVisitante.agregarVisitante( visitante );
89
```

Figura 25.22 La JavaServer Page **inicioLibroVisitantes.jsp** permite al usuario enviar un nombre de pila, apellido paterno y dirección de e-mail para colocarlas en el libro de visitantes. (Parte 2 de 3.)

http://librosysolucionarios.net

```
90      %> <%-- terminar scriptlet para insertar acción jsp:forward --%>
91
92            <%-- reenviar para mostrar el contenido del libro de visitantes --%>
93            <jsp:forward page = "vistaLibroVisitantes.jsp" />
94
95      <% // continuar scriptlet
96
97            } // fin de instrucción else
98
99      %> <%-- fin de scriptlet --%>
100  </body>
101
102  </html>
```

Figura 25.22 La JavaServer Page **inicioLibroVisitantes.jsp** permite al usuario enviar un nombre de pila, apellido paterno y dirección de e-mail para colocarlas en el libro de visitantes. (Parte 3 de 3.)

En la línea 8 de **inicioLibroVisitantes.jsp** se utiliza la *directiva* **page**, la cual define la información que está globalmente disponible en una JSP. Las directivas están delimitadas por **<%@** y **%>**. En este caso, el *atributo* **errorPage** de la directiva **page** se establece en **paginaErrorLibroVisitantes.jsp** (figura 25.24), indicando que todas las excepciones que no se atrapen se reenviarán a **paginaErrorLibroVisitantes.jsp** para su procesamiento.

En las líneas 11 a 14 se definen dos acciones **<jsp:useBean>**. En las líneas 11 y 12 se crea una instancia de un objeto **BeanInvitado** llamado **invitado**. Este bean tiene alcance tipo **page** (existe para usarse solamente en esta página). En las 13 y 14 se crea una instancia de un objeto **BeanDatosInvitado** llamado **datosInvitado**. Este bean tiene alcance tipo **request** (existe para usarse en esta página y en cualquier otra que ayude a procesar una sola petición de un cliente). Por lo tanto, cuando **inicioLibroVisitantes.jsp** reenvía una petición hacia **vistaLibroVisitantes.jsp**, sigue estando disponible la misma instancia de **BeanDatosVisitante** para usarse en **vistaLibroVisitantes.jsp**.

En la línea 36 se muestra cómo configurar las propiedades del objeto **BeanVisitante** llamado **visitante** mediante los valores de los parámetros de la petición. Los elementos **input** de las líneas 55, 63 y 71 tienen los mismos nombres que las propiedades de **BeanVisitante**. Por lo tanto, utilizamos la habilidad de la acción **<jsp:setProperty>** para relacionar los parámetros de la petición con las propiedades del bean, especificando el valor "*" para el atributo **property**. En la línea 36 también se pueden establecer las propiedades individualmente, utilizando las siguientes líneas:

```
<jsp:setProperty name = "invitado" property = "nombrePila"
   param = "nombrePila" />

<jsp:setProperty name = "invitado" property = "apellidoPaterno"
   param = "apellidoPaterno" />

<jsp:setProperty name = "invitado" property = "email"
   param = "email" />
```

Si los parámetros de la petición tuvieran nombres distintos a los de las propiedades de **BeanVisitante**, podría cambiarse el atributo **param** en cada una de las acciones **<jsp:setProperty>** anteriores al nombre del parámetro de petición apropiado.

La JavaServer Page **vistaLibroVisitantes.jsp** (figura 25.23) muestra un documento de XHTML que contiene las entradas en el libro de visitantes en formato tabular. En las líneas 8 a 10 se definen tres directivas **page**. En la línea 8 se especifica que la página de error para esta JSP es **paginaErrorLibroVisitantes.jsp**. En la línea 9 se indica que en esta JSP se utilizan las clases del paquete **java.util**, y en la línea 10 se indica que también se utilizan las clases de nuestro paquete **com.deitel.cpej5.jsp.beans**.

En las líneas 13 y 14 se especifica una acción **<jsp:useBean>** que declara una referencia a un objeto **BeanDatosVisitante**. Si ya existe un objeto **BeanDatosVisitante**, la acción devuelve una referencia al objeto existente. En caso contrario, la acción crea un objeto **BeanDatosVisitante** para usarlo en esta JSP. En las líneas 50 a 59 se define un scriptlet que obtiene la lista de visitantes del objeto **BeanDatosVisitante** y se comienza un ciclo para enviar las entradas a la salida. En las líneas 61 a 70 se combina el

```
1   <?xml version = "1.0"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. 25.23: vistaLibroVisitantes.jsp -->
6
7   <%-- configuración de página --%>
8   <%@ page errorPage = "paginaErrorLibroVisitantes.jsp" %>
9   <%@ page import = "java.util.*" %>
10  <%@ page import = "com.deitel.cpej5.jsp.beans.*" %>
11
12  <%-- objeto BeanDatosInvitado para obtener lista de visitantes --%>
13  <jsp:useBean id = "datosVisitante" scope = "request"
14     class = "com.deitel.cpej5.jsp.beans.BeanDatosVisitante" />
15
16  <html xmlns = "http://www.w3.org/1999/xhtml">
17
18     <head>
19        <title>Lista de visitantes</title>
20
21        <style type = "text/css">
22           body {
23              font-family: tahoma, helvetica, arial, sans-serif;
24           }
25
26           table, tr, td, th {
27              text-align: center;
28              font-size: .9em;
29              border: 3px groove;
30              padding: 5px;
31              background-color: #dddddd;
32           }
33        </style>
34     </head>
35
36     <body>
37        <p style = "font-size: 2em;">Lista de visitantes</p>
38
39        <table>
40           <thead>
41              <tr>
42                 <th style = "width: 100px;">Apellido paterno</th>
43                 <th style = "width: 100px;">Nombre de pila</th>
44                 <th style = "width: 200px;">E-mail</th>
45              </tr>
46           </thead>
47
```

Figura 25.23 La JavaServer Page **vistaLibroVisitantes.jsp** muestra el contenido del libro de visitantes. (Parte 1 de 2.)

http://librosysolucionarios.net

```
48          <tbody>
49
50          <% // iniciar scriptlet
51
52             List listaVisitantes = datosVisitante.getListaVisitantes();
53             Iterator iteradorListaVisitantes = listaVisitantes.iterator();
54             BeanVisitante visitante;
55
56             while ( iteradorListaVisitantes.hasNext() ) {
57                visitante = ( BeanVisitante ) iteradorListaVisitantes.next();
58
59          %> <%-- terminar scriptlet; insertar datos de plantilla fija --%>
60
61                <tr>
62                   <td><%= visitante.getApellidoPaterno() %></td>
63
64                   <td><%= visitante.getNombrePila() %></td>
65
66                   <td>
67                      <a href = "mailto:<%= visitante.getEmail() %>">
68                         <%= visitante.getEmail() %></a>
69                   </td>
70                </tr>
71
72          <% // continuar scriptlet
73
74             } // fin de instrucción while
75
76          %> <%-- fin de scriptlet --%>
77
78          </tbody>
79       </table>
80    </body>
81
82 </html>
```

Figura 25.23 La JavaServer Page **vistaLibroVisitantes.jsp** muestra el contenido del libro de visitantes. (Parte 2 de 2.)

texto de plantilla fija con expresiones de JSP para crear filas en la tabla de datos del libro de visitantes que se va a mostrar en el cliente. El scriptlet de las líneas 72 a 76 termina el ciclo.

La JavaServer Page **paginaErrorLibroVisitantes.jsp** (figura 25.24) envía a la salida un documento de XHTML que contiene un mensaje de error, con base en el tipo de excepción que provocó la invocación de esta página de error. En las líneas 8 a 10 se definen varias directivas **page**. En la línea 8 se establece

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 25.24: paginaErrorLibroVisitantes.jsp -->
6
7  <%-- configuración de página --%>
8  <%@ page isErrorPage = "true" %>
```

Figura 25.24 La JavaServer Page **paginaErrorLibroVisitantes.jsp** responde a las excepciones en **inicioLibroVisitantes.jsp** y **vistaLibroVisitantes.jsp**. (Parte 1 de 2.)

http://librosysolucionarios.net


```
9  <%@ page import = "java.util.*" %>
10 <%@ page import = "java.sql.*" %>
11
12 <html xmlns = "http://www.w3.org/1999/xhtml">
13
14    <head>
15       <title>¡Error!</title>
16
17       <style type = "text/css">
18          .rojoGrande {
19             font-size: 2em;
20             color: red;
21             font-weight: bold;
22          }
23       </style>
24    </head>
25
26    <body>
27       <p class = "rojoGrande">
28
29       <% // scriptlet para determinar el tipo de la excepción
30          // y enviar a la salida el comienzo del mensaje de error
31          if ( exception instanceof SQLException )
32       %>
33
34             Una excepción SQLException
35
36       <%
37          else if ( exception instanceof ClassNotFoundException )
38       %>
39
40             Una excepción ClassNotFoundException
41
42       <%
43          else
44       %>
45
46             Una excepción
47
48       <%-- terminar scriptlet para insertar datos de plantilla fija --%>
49
50          <%-- continuar enviando a la salida el mensaje de error --%>
51          ocurrió al interactuar con la base de datos del libro de visitantes.
52       </p>
53
54       <p class = "rojoGrande">
55          El mensaje de error fue:<br />
56          <%= exception.getMessage() %>
57       </p>
58
59       <p class = "rojoGrande">Por favor intente de nuevo más tarde</p>
60    </body>
61
62 </html>
```

Figura 25.24 La JavaServer Page **paginaErrorLibroVisitantes.jsp** responde a las excepciones en **inicioLibroVisitantes.jsp** y **vistaLibroVisitantes.jsp**. (Parte 2 de 2.)

el atributo `isErrorPage` de la directiva `page`. Al establecer este atributo en `true`, la página JSP se convierte en una página de error y permite el acceso al objeto implícito de JSP `exception`, el cual hace referencia a un objeto excepción que indica el problema que ocurrió.



Error común de programación 25.10

El objeto implícito de JSP `exception` puede utilizarse solamente en páginas de error. Si se utiliza este objeto en otras JSPs se produce un error en tiempo de traducción.

En las líneas 29 a 46 se definen scriptlets que determinan el tipo de excepción que ocurrió y se empieza a enviar a la salida un mensaje de error apropiado, mediante datos de plantilla fija. El verdadero mensaje de error de la excepción se envía en la línea 56.

En la figura 25.25 se muestran ejemplos de interacciones entre el usuario y las JSPs en el ejemplo del libro de visitantes. En las primeras dos filas de salida, dos usuarios separados escribieron su nombre de pila, apellido paterno y dirección de e-mail. En cada caso se devuelve el contenido actual del libro de visitante y se muestra en pantalla para el usuario. En la interacción final, un tercer usuario especificó una dirección de e-mail que ya existía en la base de datos. La dirección de e-mail es la clave principal en la tabla `visitantes` de la base de datos `librovisitantes`, por lo que sus valores deben ser únicos. Por lo tanto, la base de datos evita que se inserte el nuevo registro y se produce una excepción. La excepción se reenvía hacia `paginaError-LibroVisitantes.jsp` para su procesamiento, que produce como resultado la última captura de pantalla.

Para probar el libro de visitantes en Tomcat, copie los archivos `inicioLibroVisitantes.jsp`, `vistaLibroVisitantes.jsp` y `paginaErrorLibroVisitantes.jsp` en el directorio `jsp` que creó en la sección 25.3. Copie los archivos `BeanVisitante.class` y `BeanDatosVisitante.class` en el directorio `WEB-INF\classes` de la aplicación Web `cpej5` en Tomcat. (*Nota:* Este ejemplo funcionará

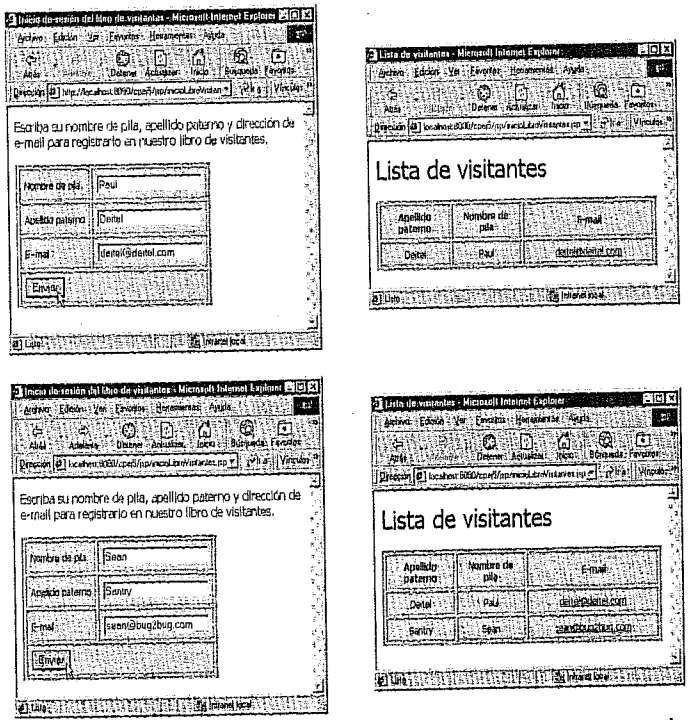


Figura 25.25 Ventanas con los resultados de ejemplo del libro de visitantes de JSP. (Parte 1 de 2.)

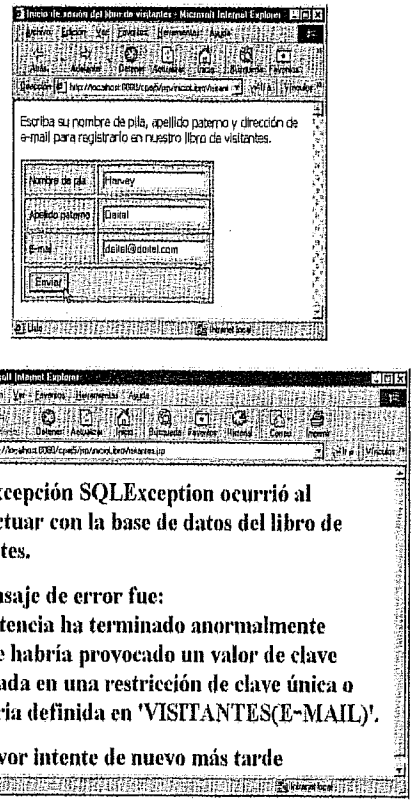


Figura 25.25 Ventanas con los resultados de ejemplo del libro de visitantes de JSP. (Parte 2 de 2.)

solamente si se define la estructura de directorios del paquete apropiada para `BeanVisitante` y `BeanDatos-Visitante`, en el directorio `classes`. Estas clases están definidas en el paquete `com.deitel.cpej5.-jsp.beans`.) Cree la base de datos ejecutando la secuencia de comandos `librovisitantes.sql`. Abra su navegador Web y escriba el siguiente URL para probar la página `inicioLibroVisitantes.jsp`:

```
http://localhost:8080/cpej5/jsp/inicioLibroVisitantes.jsp
```

25.9 Recursos en Internet y World Wide Web

`java.sun.com/products/jsp`
La página inicial para obtener información acerca de la tecnología JavaServer Pages en el sitio sobre Java de Sun Microsystems.

`java.sun.com/products/servlet`
La página inicial para obtener información acerca de los servlets en el sitio sobre Java de Sun Microsystems.

`java.sun.com/j2ee`
La página inicial de Java 2 Enterprise Edition en el sitio sobre Java de Sun Microsystems.

`www.w3.org`
La página inicial del Consorcio World Wide Web. Este sitio proporciona información acerca de los estándares de Internet y Web actuales y en desarrollo, como XHTML, XML y CSS.

http://librosysolucionarios.net

`jsptags.com`
Este sitio incluye tutoriales, bibliotecas de marcas, software y demás recursos para los programadores de JSP.

`jspinsider.com`
Este sitio sobre programación Web se concentra en ofrecer recursos para los programadores de JSP. Incluye software, tutoriales, artículos, código de ejemplo, referencias y vínculos hacia otros recursos de programación Web y JSP.

RESUMEN

- Las JavaServer Pages (JSPs) son una extensión de la tecnología de los servlets.
- Las JavaServer Pages permiten a los programadores de aplicaciones Web crear contenido dinámico mediante la reutilización de componentes predefinidos, y mediante la interacción con los componentes utilizando secuencias de comandos del lado del servidor.
- Los programadores de JSP pueden crear bibliotecas de marcas personalizadas que permitan a los diseñadores Web que no están familiarizados con Java mejorar las páginas Web, con poderosas herramientas de procesamiento y contenido dinámico.
- Las clases e interfaces específicas para la programación con JavaServer Pages se encuentran en los paquetes `javax.servlet.jsp` y `javax.servlet.jsp.tagext`.
- La especificación JavaServer Pages 1.1 puede descargarse de `java.sun.com/products/jsp/download.html`.
- Hay cuatro componentes clave para las JSPs: directivas, acciones, scriptlets y bibliotecas de marcas.
- Las directivas especifican la información global que no está asociada con la petición de una JSP específica.
- Las acciones encapsulan la funcionalidad en marcas predefinidas que los programadores pueden incrustar en una JSP.
- Los scriptlets, o elementos de secuencias de comandos, permiten a los programadores insertar código de Java que interactúe con los componentes de una JSP (y posiblemente con otros componentes de la aplicación Web) para realizar el procesamiento de las peticiones.
- Las bibliotecas de marcas son parte del mecanismo de extensión de marcas que permite a los programadores crear nuevas marcas que encapsulen la compleja funcionalidad de Java.
- Las JSPs generalmente incluyen marcado de XHTML o XML. A dicho marcado se le conoce como datos de plantilla fija o texto de plantilla fija.
- Los programadores tienden a utilizar JSPs cuando la mayoría del contenido que se envía al cliente está compuesto por datos de plantilla fija y sólo una pequeña porción del contenido se genera en forma dinámica mediante código de Java.
- Los programadores utilizan servlets cuando una pequeña porción del contenido está compuesta por datos de plantilla fija.
- Las JSPs normalmente se ejecutan como parte de un servidor Web. A este servidor se le conoce comúnmente como el contenedor de JSPs.
- Cuando un servidor habilitado para usar JSPs recibe la primera petición de una JSP, el contenedor de JSPs traduce esa JSP en un servlet de Java que maneja la petición actual y las peticiones futuras de esa JSP.
- El contenedor de JSPs coloca las instrucciones de Java que implementan la petición de una JSP en el método `_jspService`, en tiempo de traducción.
- El mecanismo de petición/respuesta y el ciclo de vida de una JSP son iguales que los de un servlet.
- Las JSPs pueden definir métodos llamados `jspInit` y `jspDestroy`, los cuales se invocan cuando el contenedor inicializa una JSP y cuando termina una JSP, respectivamente.
- Las expresiones de JSP están delimitadas por `<%=` y `%>`. Dichas expresiones son convertidas en objetos `String` por el contenedor de JSPs, y se envían como parte de la respuesta.
- El elemento `meta` de XHTML puede establecer un intervalo de actualización para un documento que se carga en un navegador. Esto hace que el navegador pida el documento repetidamente, según el intervalo especificado en segundos.
- La primera vez que se invoca la JSP en Tomcat hay un ligero retraso a medida que Tomcat traduce la JSP en un servlet e invoca a este servlet para que responda a su petición.
- Los objetos implícitos proporcionan a los programadores herramientas de servlets, dentro del contexto de una JavaServer Page.
- Los objetos implícitos tienen cuatro alcances: aplicación (`application`), página (`page`), petición (`request`) y sesión (`session`).

- Los objetos con alcance de aplicación forman parte de la aplicación contenedora de servlets y JSPs. Los objetos con alcance de página existen solamente como parte de la página en la que se utilizan. Cada página tiene sus propias instancias de los objetos implícitos con alcance de página. Los objetos con alcance de petición existen durante el tiempo de vida de la petición. Los objetos con alcance de petición quedan fuera de alcance cuando se completa el procesamiento de la petición mediante una respuesta al cliente. Los objetos con alcance de sesión existen durante toda la sesión de navegación del cliente.
- Los componentes de las secuencias de comandos de JSP incluyen scriptlets, comentarios, expresiones, declaraciones y secuencias de escape.
- Los scriptlets son bloques de código delimitado por `<%` y `%>`. Contienen instrucciones de Java que se colocan en el método `_jspService` cuando el contenedor traduce una JSP en un servlet.
- Los comentarios de JSP están delimitados por los caracteres `<%--` y `--%>`. Los comentarios de XHTML están delimitados por los caracteres `<!--` y `-->`. Dentro de los scriptlets se pueden utilizar los comentarios de fin de línea de Java (`//`) y los comentarios tradicionales (delimitados por `/*` y `*/`).
- Los comentarios de JSP y los comentarios de lenguaje de secuencias de comandos se ignoran y no aparecen en la respuesta.
- Una expresión de JSP delimitada por los caracteres `<%=` y `%>` contiene una expresión de Java que se evalúa cuando un cliente pide la JSP que contiene esa expresión. El contenedor convierte el resultado de una expresión JSP en un objeto `String` y después envía ese objeto `String` como parte de la respuesta al cliente.
- Las declaraciones (delimitadas por los caracteres `<%!` y `%>`) permiten a un programador de JSPs definir variables y métodos. Las variables se convierten en variables de instancia de la clase que representa a la JSP traducida. De manera similar, los métodos se convierten en miembros de la clase que representa a la JSP traducida.
- Los caracteres o secuencias de caracteres especiales que utiliza normalmente el contenedor de JSPs para delimitar código de JSP pueden incluirse en una JSP como caracteres literales en elementos de secuencias de comandos, datos de plantilla fija y valores de atributos, mediante el uso de secuencias de escape.
- Las acciones estándar de JSP proporcionan a los que implementan JSPs el acceso a varias de las tareas más comunes que se realizan en una JSP. Los contenedores de JSPs procesan las acciones en tiempo de petición.
- Las JavaServer Pages soportan dos mecanismos de inclusión: la acción `<jsp:include>` y la directiva `include`.
- La acción `<jsp:include>` permite incluir contenido dinámico en una JavaServer Page. Si el recurso incluido cambia entre peticiones, la siguiente petición a la JSP que contenga la acción `<jsp:include>` incluirá el nuevo contenido del recurso.
- La directiva `include` se procesa una vez (en tiempo de traducción de la JSP) y hace que su contenido se copie en la JSP. Si el recurso incluido cambia, el nuevo contenido no se reflejará en la JSP que utilizó la directiva `include`, a menos que se vuelva a compilar esa JSP.
- La acción `<jsp:forward>` permite a una JSP reenviar el procesamiento de una petición a un recurso distinto. El procesamiento de la petición por parte de la JSP original termina tan pronto como la JSP reenvía la petición.
- La acción `<jsp:param>` especifica los pares nombre/valor de información que se pasan a las acciones `include`, `forward` y `plugin`. Cada acción `<jsp:param>` tiene dos atributos requeridos: `name` y `value`. Si una acción `param` especifica un parámetro que ya existe en la petición, el nuevo valor para ese parámetro tiene precedencia sobre el valor original. Todos los valores para ese parámetro pueden obtenerse mediante el método `getParameterValues` del objeto implícito de JSP `request`, el cual devuelve un arreglo de objetos `String`.
- La acción `<jsp:useBean>` permite a una JSP manipular un objeto de Java. Esta acción puede utilizarse para crear un objeto de Java que pueda usarse en la JSP, o para localizar un objeto existente.
- Al igual que los objetos implícitos de JSP, los objetos que se especifican con la acción `<jsp:useBean>` tienen alcance tipo `page`, `request`, `session` o `application`, el cual indica en qué parte de una aplicación Web pueden utilizarse.
- La acción `<jsp:getProperty>` obtiene el valor de una propiedad de un JavaBean. La acción `<jsp:getProperty>` tiene dos atributos (`name` y `property`) que especifican el objeto bean que se va a manipular y la propiedad que se va a obtener.
- Los valores de las propiedades de un JavaBean pueden establecerse con la acción `<jsp:setProperty>`, la cual es particularmente útil para asociar los valores de los parámetros de una petición con las propiedades del JavaBean. Los parámetros de petición pueden utilizarse para establecer propiedades que tengan los tipos primitivos `boolean`, `byte`,

http://librosysolucionarios.net

`char`, `int`, `long`, `float` y `double`, y los tipos `String`, `Boolean`, `Byte`, `Character`, `Integer`, `Long`, `Float` y `Double` de `java.lang`.

- La directiva `page` define la información que está globalmente disponible en una JSP. Las directivas están delimitadas por `<%@` y `%>`. El atributo `errorPage` de la directiva `page` indica hacia dónde se reenvían toda las excepciones no atrapadas para su procesamiento.
- La acción `<jsp:setProperty>` tiene la habilidad de relacionar los parámetros de una petición con las propiedades del mismo nombre en un bean, especificando `"*"` para el atributo `property`.
- El atributo `import` de la directiva `page` permite a los programadores especificar las clases y paquetes de Java que se van a utilizar dentro del contexto de una JSP.
- Si el atributo `isErrorPage` de la directiva `page` se establece en `true`, entonces la JSP es una página de error. Esta condición permite el acceso al objeto implícito de JSP `exception`, el cual hace referencia a un objeto excepción que indica el problema que ocurrió.
- Las directivas son mensajes para el contenedor de JSPs, que permiten al programador especificar configuraciones de página (como la página de error), incluir contenido de otros recursos y especificar bibliotecas de marcas personalizadas que pueden utilizarse en una JSP. Las directivas se procesan al momento en que una JSP se traduce en un servlet y se compila. Por lo tanto, las directivas no producen ninguna salida inmediata.
- La directiva `page` especifica las configuraciones globales para una JSP en el contenedor de JSPs. Puede haber muchas directivas `page`, siempre y cuando sólo haya una ocurrencia de cada atributo. La excepción a esta regla es el atributo `import`, el cual puede usarse repetidamente para importar paquetes de Java.

TERMINOLOGÍA

`%\>`, secuencia de escape para `%>`
`_jspService`, método
`<!--` y `-->`, delimitadores de comentarios de XHTML
`<%--` y `--%>`, delimitadores de comentarios de JSP
`<%` y `%>`, delimitadores de scriptlets
`<%!` y `%>`, delimitadores de declaraciones
`<%@` y `%>`, delimitadores de directivas
`<%=` y `<%=`, delimitadores de expresiones de JSP
`<\%`, secuencia de escape para `<%`
`<jsp:forward>`, acción
`<jsp:getProperty>`, acción
`<jsp:include>`, acción
`<jsp:param>`, acción
`<jsp:setProperty>`, acción
`<jsp:useBean>`, acción
acción
acciones estándar
alcance de página
alcance de petición
alcance del bean
alcances de objetos implícitos
`autoFlush`, atributo de la directiva `page`
`beanName`, atributo de la acción `<jsp:useBean>`
`buffer`, atributo de la directiva `page`
`class`, atributo de la acción `<jsp:useBean>`
comentario
`config`, objeto implícito
contenedor
contenido dinámico
`contentType`, atributo de la directiva `page`
datos de plantilla fija
declaración
directiva
elementos de secuencias de comandos
error en tiempo de petición
error en tiempo de traducción
`errorPage`, atributo de la directiva `page`
especificar los atributos de una marca personalizada
expresión
`extends`, atributo de la directiva `page`
`file`, atributo de la directiva `include`
`flush`, atributo de la acción `<jsp:include>`
`getParameterValues`, método del objeto `request`
`id`, atributo de la acción `<jsp:useBean>`
`import`, atributo de la directiva `page`
`include`, directiva
incluir recursos
inclusión en tiempo de traducción
`info`, atributo de la directiva `page`
intervalo de actualización
`isErrorPage`, atributo de la directiva `page`
`isThreadSafe`, atributo de la directiva `page`
JavaServer Pages (JSPs)
JavaServer Pages 1.1, especificación
`javax.servlet.jsp`, paquete
`javax.servlet.jsp.tagext`, paquete
`jspDestroy`, método
`jspInit`, método
`JspWriter` (paquete `javax.servlet.jsp`)
`language`, atributo de la directiva `page`
`meta`, elemento
`name`, atributo de `<jsp:param>`
`name`, atributo de `<jsp:setProperty>`
objeto implícito
`out`, objeto implícito
`page`, atributo de `<jsp:forward>`
`page`, atributo de `<jsp:include>`
`page`, directiva
`page`, objeto implícito
página de error
`param`, atributo de `<jsp:setProperty>`
pares nombre/valor
`property`, atributo de `<jsp:setProperty>`
redes cliente-servidor
reenviar peticiones
relacionar parámetros de petición
`request`, objeto implícito
`response`, objeto implícito
`scope`, atributo de `<jsp:useBean>`
scriptlet
secuencia de escape
texto de plantilla fija
`type`, atributo de `<jsp:plugin>`
`type`, atributo de `<jsp:useBean>`
`value`, atributo de `<jsp:param>`
`value`, atributo de `<jsp:setProperty>`

EJERCICIOS DE AUTOEVALUACIÓN

25.1 Complete las siguientes oraciones:

a) La acción _____ tiene la habilidad de relacionar los parámetros de una petición con las propiedades del mismo nombre en un bean, especificando `"*"` para el atributo `property`.

b) Hay cuatro componentes clave de las JSPs: _____, _____, _____ y _____.

c) Los objetos implícitos tienen cuatro alcances: _____, _____, _____ y _____.

d) La directiva _____ se procesa una vez (en tiempo de traducción de la JSP) y hace que su contenido se copie en la JSP.

e) Las clases e interfaces específicas para la programación con JavaServer Pages se encuentran en los paquetes _____ y _____.

f) Las JSPs se ejecutan normalmente como parte de un servidor Web, al cual se le conoce como _____.

g) Los componentes de las secuencias de comandos de JSP incluyen a _____, _____, _____, _____ y _____.

25.2 Conteste con verdadero o falso a cada una de las siguientes proposiciones; en caso de ser falso, explique por qué.

a) Un objeto en un alcance de página existe en cada una de las JSPs de una aplicación Web específica.

b) Las directivas especifican la información global que no está asociada con la petición de una JSP específica.

c) La acción `<jsp:include>` se evalúa una vez, en tiempo de traducción de la página.

d) Al igual que los comentarios de XHTML, los comentarios de JSP y de lenguaje de secuencias de comandos aparecen en la respuesta al cliente.

e) Los objetos con alcance de aplicación forman parte de una aplicación Web específica.

f) Cada página tiene sus propias instancias de los objetos implícitos con alcance de página.

g) La acción `<jsp:setProperty>` tiene la habilidad de relacionar los parámetros de una petición con las propiedades del mismo nombre en un bean, especificando `"*"` para el atributo `property`.

h) Los objetos con alcance de sesión existen durante toda la sesión de navegación del cliente.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

25.1 a) `<jsp:setProperty>`. b) Directivas, acciones, scriptlets, bibliotecas de marca. c) Aplicación, página, petición y sesión. d) `include`. e) `javax.servlet.jsp`, `javax.servlet.jsp.tagext`. f) Contenedor de JSPs. g) Scriptlets, comentarios, expresiones, declaraciones, secuencias de escape.

25.2 a) Falso. Los objetos con alcance de página existen solamente como parte de la página en la que se utilizan. b) Verdadero. c) Falso. La acción `<jsp:include>` permite incluir contenido dinámico en una JavaServer Page. d) Falso. Los comentarios de JSP y los comentarios de lenguaje de secuencias de comandos se ignoran y no aparecen en la respuesta. e) Falso. Los objetos con alcance de aplicación forman parte de la aplicación contenedora de JSPs. f) Verdadero. g) Verdadero. h) Verdadero.

EJERCICIOS

25.3 Escriba una página JSP para mostrar la cadena "¡Hola, mundo!" diez veces.

25.4 Modifique el ejercicio 24.4 para que se ejecute como una página JSP.

25.5 Realice cambios en la figura 25.15 para permitir que los usuarios seleccionen la imagen. Utilice una expresión de JSP en vez de la marca de JSP `getProperty`.

http://librosysolucionarios.net

25.6 Cree una libreta de direcciones basada en JSP y JSBC. Utilice el ejemplo del libro de visitantes de las figura 25.20 a 25.24 como guía. Su libreta de direcciones deberá permitir al usuario escribir entradas, borrarlas y realizar búsquedas.

25.7 Reimplemente la aplicación Web de la figura 24.20 (encuesta sobre animales favoritos) utilizando JSPs.

25.8 Modifique su solución al ejercicio 25.7 para permitir al usuario ver los resultados de la encuesta sin tener que responder a la misma.

A

Tabla de precedencia de los operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo.

Operador	Descripción	Asociatividad
`++`	unario de postincremento	de derecha a izquierda
`--`	unario de postdecremento	
`++`	unario de preincremento	de derecha a izquierda
`--`	unario de predecremento	
`+`	unario de suma	
`-`	unario de resta	
`!`	unario de negación lógica	
`~`	unario de complemento a nivel de bits	
(*tipo*)	unario de conversión	
`*`	multiplicación	de izquierda a derecha
`/`	división	
`%`	residuo	
`+`	suma o concatenación de cadenas	de izquierda a derecha
`-`	resta	
`<<`	desplazamiento a la izquierda	de izquierda a derecha
`>>`	desplazamiento a la derecha con signo	
`>>>`	desplazamiento a la derecha sin signo	
`<`	menor que	de izquierda a derecha
`<=`	menor o igual que	
`>`	mayor que	
`>=`	mayor o igual que	
`instanceof`	comparación de tipos	

Figura A.1 Tabla de precedencia de los operadores. (Parte 1 de 2.)

http://librosysolucionarios.net

Operador	Descripción	Asociatividad
==	es igual a	de izquierda a derecha
!=	no es igual a	
&	AND a nivel de bits	de izquierda a derecha
	AND lógico booleano	
^	OR excluyente a nivel de bits	de izquierda a derecha
	OR excluyente lógico booleano	
|	OR incluyente a nivel de bits	de izquierda a derecha
	OR incluyente lógico booleano	
&&	AND condicional	de izquierda a derecha
||	OR condicional	de izquierda a derecha
?:	condicional	de derecha a izquierda
=	asignación	de derecha a izquierda
+=	asignación, suma	
-=	asignación, resta	
*=	asignación, multiplicación	
/=	asignación, división	
%=	asignación, residuo	
&=	asignación, AND a nivel de bits	
^=	asignación, OR excluyente a nivel de bits	
|=	asignación, OR incluyente a nivel de bits	
<<=	asignación, desplazamiento a la izquierda a nivel de bits	
>>=	asignación, desplazamiento a la derecha a nivel de bits con signo	
>>>=	asignación, desplazamiento a la derecha a nivel de bits sin signo	

Figura A.1 Tabla de precedencia de los operadores. (Parte 2 de 2.)

B

Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Figura B.1 El conjunto de caracteres ASCII.

Los dígitos a la izquierda de la tabla son los dígitos izquierdos (decenas) del equivalente decimal (0-127) del código de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos (unidades) del código de caracteres. Por ejemplo, el código de carácter para la "F" es 70, mientras que para el "&" es 38.

La mayoría de los usuarios de este libro estarán interesados en el conjunto de caracteres ASCII utilizado para representar los caracteres del idioma español en muchas computadoras. El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres Unicode utilizado por Java para representar caracteres de la mayoría de los lenguajes existentes en el mundo. Para obtener más información acerca del conjunto de caracteres Unicode, vea el apéndice G.

http://librosysolucionarios.net

http://libreria-universitaria.blogspot.com

Apéndices

C a G

El material correspondiente a los apéndices C a G podrá encontrarlo en el CD-ROM que acompaña a este libro, en archivos en formato PDF totalmente en español.

Apéndice C	**Sistemas numéricos**
Apéndice D	**Eventos e interfaces de escucha del elevador**
Apéndice E	**Modelo del elevador**
Apéndice F	**La vista del Elevador (en CD)**
Apéndice G	**Unicode®**

Bibliografía

Recursos de Sun Microsystems

Block, J., "Tutorial: Collections",
`java.sun.com/docs/books/tutorial/collections/index.html`

Fisher, M., "The JDBC Tutorial and Reference: Second Edition" (extracto del capítulo 3),
`java.sun.com/ developer/Books/JDBCTutorial/index.html`

Gosling, B., B. Joy, G. Steele y G. Bracha, "The Java Language Specification: Second Edition",
`java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`

Lindholm, T., y F. Yellin, "The Java Virtual Machine Specification: Second Edition",
`java.sun.com/ docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

Meloan, M. D., "The Science Of Java Sound",
`java.sun.com/developer/technicalArticles/ Media/JavaSoundAPI/index.html`

Sun Microsystems, "Java Technology and XML",
`java.sun.com/xml`

Sun Microsystems, "Java 2D API Specifications and Tutorials",
`java.sun.com/products/java-media/2D`

Sun Microsystems, "Java Servlet Technology",
`java.sun.com/products/servlet/index.html`

Sun Microsystems, "JavaServer Pages: Dynamically Generated Web Content",
`java.sun.com/products/jsp/index.html`

Sun Microsystems, "The Collections Framework Overview",
`java.sun.com/j2se/1.4.2/docs/guide/collections/overview.html`

Sun Microsystems, "JDBC Data Access API",
`java.sun.com/products/jdbc/overview.html`

Sun Microsystems, "Lesson: Getting Started with Swing",
`java.sun.com/docs/books/tutorial/uiswing/start/index.html`

Sun Microsystems, "New to Java Training Center",
`java.sun.com/learning/new2java/index.html`

Sun Microsystems, "The Java Tutorial",
`java.sun.com/docs/books/tutorial`

http://librosysolucionarios.net

Sun Microsystems, "Lesson: Swing Features and Concepts", `java.sun.com/docs/books/tutorial/uiswing/overview/index.html`

Sun Microsystems, "Lesson: Using Swing Components", `java.sun.com/docs/books/tutorial/uiswing/components/index.html`

Sun Microsystems, "Lesson: Laying Out Components within a Container", `java.sun.com/docs/books/tutorial/uiswing/layout/index.html`

Otros recursos

Arnold, K., J. Gosling y D. Holmes. *The Java™ Programming Language: Third Edition*. Reading, MA: Addison-Wesley, 2000.

Barker, J. *Beginning Java Objects: From Concepts to Code*. Birmingham: Wrox Press, 2000.

Bennet, S., J. Skelton, y K. Lunn. *Schaum's Outline of UML*. Nueva York, NY: McGraw Hill, 2001.

Berg, D. y Fritzinger, J.S., *Advanced Techniques for Java Developers*. Nueva York, NY: John Wiley & Sons, Inc., 1998.

Bloch, J. *Effective Java™ Programming Language Guide*. Reading, MA: Addison-Wesley, 2001.

Booch, G. *Object-Oriented Analysis and Design with Applications*. Reading, MA: Addison-Wesley, 1994.

Booch, G., J. Rumbaugh, e I. Jacobson. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall PTR, 2000.

Brodsky, S. y T. Grose. *Mastering XMI: Java Programming XMI, XML, and UML*. Nueva York, NY: John Wiley & Sons, 2002.

Brogden, B., *Java 2 Exam Cram, Second Edition*, Scottsdale, AZ: The Coriolis Group, 2001.

Cheesman, J., y J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software (The Component Software Series)*. Reading, MA: Addison-Wesley, 2000.

Daconta, M. C., E. Monk, J. P. Keller, y K. Bohnenberger. *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs*. Nueva York, NY: John Wiley & Sons, 2000.

Eckel, B. *Thinking In Java: 2nd Edition*. Upper Saddle River, NJ: Prentice Hall, 2000.

Flanagan, D. *Java Examples In A Nutshell*. Sabastopol, CA: O'Reilly and Associates, 2000.

Folwer, M. y K. Scott. *UML Distilled Second Edition; A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1999.

Gamma, E., R. Helm, R. Johnson y J. Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

Gilbert, S., y McCarty, B., *Object-Oriented Design in Java*. Corte Madera, CA: Waite Group Press, 1998.

Grand, M. *Patterns in Java; A Catalog Reusable Design Patterns Illustrated with UML (Segunda edición)*. Nueva York, NY: John Wiley & Sons, 2002.

Horstmann, C. S. y G. Cornell. *Core Java 2: Volume 1-Fundamentals (Sexta edición)*. Upper Saddle River, NJ: Prentice Hall, 2003.

Horstmann, C. S. y G. Cornell. *Core Java 2: Volume II-Advanced Features (Quinta edición)*. Upper Saddle River, NJ; Prentice Hall, 2002.

Larman, C. *Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, NJ: Prentice Hall, 2002.

Lea, D. *Concurrent Programming in Java™ Second Edition Design Principles and Patterns*. Reading, MA: Addison-Wesley, 2000.

Lauinger, T., "Object-Oriented Software Development in Java", *Java Report*, febrero de 1999, págs. 59 a 61.

Malarvannan, M., "A Multithreaded Server in Java", *Web Techniques*, octubre de 1998, págs. 47 a 51.

Maruyama, H., A. Clark, M. Murata, M. Uramoto, K. Tamamura, Y. Nakamura, R. Neyama, K. Kisaka y S. Hada., *XML and JAVA: Developing Web Applications (Segunda edición)*, Reading, MA: Addison-Wesley Professional, 2002.

Oaks, S., "How Do I Create My Own UI Component?", Java Report, marzo/abril de 1996, págs. 63, 64.

Oaks, S., "Two Techniques for Handling Events", *Java Report*, julio/agosto de 1996, pág. 80.

Page-Jones, M. *Fundamentals of Object-Oriented Design in UML*. Reading, MA: Addison-Wesley, 1999.

Penker, M. y E. Hans-Erik. *Business Modeling with UML: Business Patterns At Work*. Nueva York, NY: John Wiley & Sons, 2000.

Roberts, Si., P. Heller, M. Ernest, y R. *et al. The Complete Java 2 Certification Study Guide (Tercera edición)*. Alameda, CA: SYBEX, 2002.

Rodrigues, L., "On JavaBeans Customization", *Java Developer's Journal*, mayo de 1999, pág. 21.

Rumbaugh, J., I. Jacobson y G. Booch. The *Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Rosenburg, D. y K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Schach, S. *Object-Oriented and Classical Software Engineering*. Nueva York, NY: McGraw Hill, 2001.

Schneider, G. y J. Winters. *Applying Use Cases*. Reading, MA: Addison-Wesley, 1998.

Scott, K. *UML Explained*. Reading, MA: Addison-Wesley, 2001.

Shirazi, J. *Java Performance Tuning*. Sebastopol, CA: O'Reilly and Associates, 2000.

Stevens, P. y R. J. Pooley. *Using UML: Software Engineering with Objects and Components Revised Edition*. Reading, MA: Addison-Wesley, 2000.

Sun Microsystems Inc. *Java™ Look and Feel Design Guidelines, Second Edition*. Reading, MA: Addison-Wesley, 2001.

Topley, K. *Core Swing: Advanced Programming*. Upper Saddle River, NJ: Prentice Hall, 2000.

Venners, B., *Inside the Java Virtual Machine*. Nueva York, NY: McGraw Hill, 2000.

Vlissides, J. *Pattern Hatching; Design Patterns Applied*. Reading, MA: Addison-Wesley, 1998.

http://librosysolucionarios.net

Índice

Símbolos

!=, 50
$, @, %, &, *, (,), -, +, ", :, ? y /, 751
%>, 1153
%>, 1155
--%>, 1155
&& (AND condicional), || (OR condicional), 167
(2pr2) + (2prh), 377
?, 1128
<!--, 1155
<%!, 1155
<%=, 1153
<, >, <=, >=, =, <>, 1081
<, 50
</applet>, 83
</html>, 83
</jsp:acción>, 1158
<=, 50
<jsp:accion />, 1159
<jsp:acción>, 1158
<jsp:forward>, 1159
<jsp:getProperty>, 1159
<jsp:include>, 1159
<jsp:param>, 1159
<jsp:plugin>, 1159
<jsp:setProperty>, 1159, 1170
<jsp:useBean>, 1159
-->, 1155
>, 50
>=, 50
∞, símbolo de infinito, 1078

A

a prueba de subprocesos, 723
a través del tiempo, envío de mensajes entre objetos, 739
abierta, atributo, 134
abrir archivos, 753
Abrir, cuadro de diálogo, 776
abrirPuerta,
  mensajes del elevador, 277
  operación del elevador, 227
absolutas, operaciones, 810
abstracción, 17
  de datos, 344
  programación orientada a objetos, 356
abstract
  clase, 512
  palabra reservada, 398, 753
AbstractButton, clase, 561
AbstractCollection, clase, 1067
AbstractList, clase, 1067
AbstractMap, clase, 1067
AbstractSequentialList, clase, 1067
AbstractSet, clase, 1067
AbstractTableModel (paquete javax.swing.table), clase, 1094
accept, método de ServerSocket, 836
acceso a nivel de paquete, 341
acciones
  de un procedimiento, 104
  estándar de JSP, 1158
acíclico, degradado, 534
acoplamiento entre clases, 667
ActionEvent, clase, 207, 323, 611, 835
ActionEventHandler, clase, 427
ActionListener
  clase, 559
  interfaz, 207
actionPerformed, método de ActionListener, 207, 323
actionPerformed, método, 417, 482, 518, 559, 613, 680, 835
activación, diagrama de secuencia, 739
actividad, diagrama, 106
actividades de los objetos, 177
actor, diagramas de caso-uso, 597
acumulador de Simpletron, 294
Adaptador, patrón de diseño, 664
add
  instrucción, 991
  método de ButtonGroup, 551
  método de la clase Container, 209
  método de List, 1050
  método de Vector, 828
addActionListener
  método, 323, 559
  método de ActionEventHandler, 427
  método de JButton, 210
addFirst, método de List, 1050
addItemListener, método, 565, 586
addLast, método de List, 1050
addLayoutComponent, método, 668
addListSelectionListener, método, 573
addMouseListener, método, 579, 617
addMouseMotionListener, método, 579, 617
addObserver, método de Observable, 667
addPoint, método de Polygon, 530
addTab, método de JTabbedPane, 643
addWindowListener
  método de JFrame, 432
  método de Window, 625
administrador de esquemas, 208, 587
ADT (tipos de datos abstractos), 303
.aif o .aiff (formato de archivo AIFF de Macintosh), 930
alcance
  de aplicación, 1153
  de la variable, 150
  de una clase, 310
algoritmo, 104, 668
  reversa, 1050
algoritmos
  mín y máx, 1050
  sort, binarySearch, reverse, shuffle, fill y copy, 1050
alias, 1124
alinear a la izquierda, 553
alinearse a la izquierda, al centro (el valor predeterminado) y a la derecha, 587
Alt, tecla, 584
altura de un tipo de letra, 521
allocate, método estático de Buffer, 812
análisis
  de requerimientos de un proyecto, 18
  y diseño orientados a objetos (A/DOO), 18
anchor, campo de GridBagConstraints, 649
anchura y altura de un subprograma, 83
AND
  a nivel de bits (&), 1015
  lógico (and), OR incluyente lógico a nivel de bits (or), 1028
  lógico booleano, 167
Anders Hejlsberg y Scott Wiltamuth, 22
ángulo
  del arco, 527
  inicial, 527
anidamiento de estructuras de control, 108
Animator, programa de ejemplo del directorio applets, 73
ANSI A, 14
ANSI C, 7
ANSI/ISO 9899:, 7
apagarLuz
  mensaje del elevador, 277
  operación de la clase Luz, 228
apariencia visual adaptable (PLAF) de Swing, 608
apariencia visual de Microsoft Windows, 608

http://librosysolucionario.net

apariencia visual metálica de Swing, 608
API, 921
  de colecciones de Java, 1002
  de Java, 188
  de Java (Interfaz de programación de aplicaciones de Java), 198
  de síntesis de voz de Java, 921
  de sonido de Java, 921
  del marco de trabajos de medios de Java (Java Media Framework), 921
API 3D de Java, 921
API Java2D, 77, 510
API JDBC™, 1074
API Preferences, 1009
APIs (Interfaces de programación de aplicaciones) de Java, 8
apilamiento de estructuras de control, 108, 172
aplazamiento indefinido, 704
aplicación
  cliente/servidor para conversar, 837
  de tres niveles, 911
aplicaciones
  cliente/servidor, 1
  de acceso instantáneo, 782
  empresariales a gran escala, 22
  Web, 1124, 1145
`append`, método de `JTextArea`, 155
applet (subprograma), 13, 23, 75, 143, 146, 157, 184, 259, 306, 309, 319, 323-324, 326, 334, 353-354, 422, 427, 443, 449, 510, 511, 512, 545, 546, 828
`Applet`, clase, 79, 922
AppletContext, interfaz, 827
applets, 13, 15, 23, 54, 73, 146, 361, 417, 422, 510
appletviewer, 14
`application`, objeto implícito, 1154
apuntador de posición de archivo, 776
árbol de búsqueda binaria, 970
`Arc2D.CHORD`, 535
`Arc2D.OPEN`, 535
`Arc2D.PIE`, 535
arco, dibujar un, 527
`ArcTest`, programa de ejemplo del directorio `applets`, 73
archivo, 751
  de clase, 33
  de fichero de aplicación Web, 1124
  por lotes, 40
  secuencial, 752
  WAR, 1124
archivos
  de acceso aleatorio, 783
  de ficheros, 340
  de texto, 83
área rígida, 647
áreas
  activas, 927
  dedicadas de dibujo, 612
argumento de línea de comandos, 74
`ArithmeticException`, clase, 680
Arquitectura común de agente de solicitud de objetos (CORBA), 1112
arrastrar el ratón, 75
`ArrayIndexOutOfBoundsException`
  clase, 532
  excepción, 257
ArrayList, clase, 1044
Arrays, clase, 1039
arreglo
  de bytes envuelto, 810
  de *m* por *n*, 270
  de respaldo, 810
arreglos, 246
  bidimensionales, 270
  multidimensionales, 270
artefactos, 657
  UML, 59
asignación dinámica de memoria, 953
asList, método de Arrays, 1042
asociación
  de clases, 96
  de servlet, 1124
  de uno a uno, 1063
asociaciones
  de extensiones, 1126
  de rutas, 1126
asociatividad, 47
asterisco (`*`), 46, 490
atrapa y maneja a esa excepción, 678
atributos
  (datos) de clases, 133, 302
  de los objetos, 17
  de un objeto, 80, 83, 96
.au (formato de archivo de audio de Sun, 930
autoasociación de clases, 500
autoFlush, atributo de la directiva page, 1172
Av Pág (avance de página), 621
AWT (Abstract Windows Toolkit), 549
`AWTEvent`, clase, 554
`AWTEvent`, clase, 666

B

Babbage, Charles, 10
balanceados, árboles, 976
Banda de los cuatro, 440
bandera
  de interrupción, 703
  valor de, 118
`BarChart`, programa de ejemplo del directorio `applets`, 73
barra
  de desplazamiento, 172, 569
  de menús, 38, 548, 625
  de título, 39, 624
  diagonal inversa, 36
base de datos, 1074, 1136
  de datos relacional, 1075
BASIC (Beginner's All-Purpose Symbolic Instruction Code, Código de instrucciones simbólicas de uso general para principiantes), 22
básicos, paquetes, 36
`BasicStroke`, clase, 510, 535
`BasicStroke.CAP_ROUND`, 536
`BasicStroke.JOIN_ROUND`, 536
bean, 1159
beanName, atributo, 1167
BevelBorder.LOWERED, 894
bibliotecas de clases, 9, 343, 356
  de Java, 8, 36, 188, 198
Biblioteca de clases del Framework (FCL), 22
bibliotecas de marcas, 1150
Bienvenido.java, 13
`BigDecimal`, clase, 220
`BigInteger`, clase 220
`binarySearch`, método de `Arrays`, 1039
bit, 751
`BitSet`, clase, 998
Blaise Pascal, 10
`Blink`, programa de ejemplo del directorio `applets`, 74
Bloc de notas, 12
bloque
  de instrucciones, 112
  `try`, 638
Bloqueado, estado de un subproceso, 702
bloqueo de archivo, 811
`boolean`, `byte`, `char`, `int`, `long`, `float`, tipos primitivos, 1170
`boolean`
  palabra reservada, 586
  variables de bandera, 733
`Boolean`
  clase, 433
  tipos, 1170
borde de una ventana, 624
BorderLayout, administrador de esquemas, 831
`BorderLayout`, clase, 579, 611, 615
`BorderLayout.CENTER`, 579, 831
`BorderLayout.EAST`, 579
`BorderLayout.NORTH`, 579, 831
`BorderLayout.SOUTH`, 579
`BorderLayout.WEST`, 579
botón, 561
`Boton`, clase del elevador, 434, 539, 739
`BotonElevador`, 134
  clase del elevador, 95, 177, 277, 434
botones, 548
  de comando, casillas de verificación, botones interruptores y botones de opción, 561
  de estado, 563
  de opción, 76, 366
`botonOprimido`, mensaje del elevador, 740
`BotonPiso`, 134
  clase del elevador, 95, 177, 277, 434
`Box`, contenedor (paquete `javax.swing`), 609
`BoxLayout`, administrador de esquemas, 609
`BoxLayout.X_AXIS`, 647
`BoxLayout.Y_AXIS`, 647
branch negative, instrucción, 988
branch zero, instrucción, 988
`break`, palabra reservada, 162
búfer, 709
  circular, 722
  directo, 810
  atributo de la directiva page, 1172
`BufferedImage`, clase, 535
`BufferedImage.TYPE_INT_RGB`, 535
`BufferedInputStream`, clase, 754
`BufferedOutputStream`, clase, 754
`BufferedReader`, clase, 755
`BufferedWriter`, clase, 755
búsqueda binaria, 240
búsqueda en arreglos, 263
  binaria, 263
  lineal, 263
`Button`, clase, 549
`ButtonGroup`, clase, 566
`Byte`
  clase, 433
  tipos, 1170
`ByteArrayInputStream`, clase, 755
`ByteArrayOutputStream`, clase, 755
`ByteBuffer`, clase, 810, 897
bytes, 751

C

C++, 22
caché, 1130
cada iteración del ciclo, 146
cadena, 666
  de consulta, 1128
  de responsabilidad, 666
  vacía, 453
cadenas, 32
  de caracteres, mensaje, 32
  vacía, 53
calcular iterativamente, 218
CallableStatement, interfaz, 1104
cambiar directorios, 73
`cambiarPisos`, mensaje del elevador, 744
campo, 90
  de texto, 38
campos, 751
  de las clases, 17
canal del archivo, 811
canalizaciones, 753
`CANCEL_OPTION`, constante, 770
cantidades escalares, 259
capacidad
  de un objeto, 998
  inicial, 1001
  límite, posición y marca, 810
capacity, método de Vector, 1002
carácter
  de escape, 36
  de nueva línea (secuencia de escape \n), 155
  de palabra, 486
  de tabulación (secuencia de escape \t), 155
  separador, 760
caracteres
  de espacio en blanco, 30
  de nueva línea, 35
  especiales, 453
`CardTest`, programa de ejemplo del directorio `applets`, 74
cargador de clases, 13, 340, 893
cargar programa en memoria, 294
`case` etiquetas, 160, 161
casilla de verificación, 77
caso `default`, 160
caso(s) base, 218
casos de uso, 596
`catch`, cláusula, 638, 678
cerrar ventana, 427
`CerrarPuerta`
  mensaje del elevador, 277
  operación del elevador, 227
CGI, 1115
cíclico, degradado, 534
ciclo
  de ejecución de una instrucción, 297
  infinito, 113
`Cilindro`
  clase, 377, 390
cima, representación de un programa, 118
círculo relleno, 106, 177
`Circulo`, clase, 359, 390
`Circulo2`, clase, 359, 366
`Circulo3`, clase, 359
`Circulo4`, clase, 359, 392
clase, 177, 303
  abstracta, 390
  activa, 739
  adaptadora, 432
  autorreferenciada, 952
  Date, 1153
  de aplicación, 309
  de envoltura, 1066
    de tipo, 433
    de tipos, 135
  de Estado, 445
  ejecutable, 309
  Integer, 43
  interna anónima que extiende a una clase, 428
  interna, 391
  iteradora, 399
  `Locale`, 155
clases, 8, 188
  abstractas, 397
  adaptadoras, 579
  anidadas, 421
  base, 79
  concretas, 397
  de nivel superior, 421
  definidas por el programador, 30
  definidas por el usuario, 30
  derivadas, 79
  internas, 421
  internas anónimas, 428
  internas anónimas separadas que implementan a una interfaz, 428
  predefinidas de caracteres, 486
`Class`
  clase, 417
  palabra reservada, 30, 313, 1167
.class, archivo, 23
`ClassCastException`, clase, 417
ClassNotFoundException, clase, 1177
clave
  de registro, 751
  externa, 1077, 1079
  principal, 1075
Clave Morse, 507
cliente, 665, 827, 1113
cliente del objeto, 80
clientes delgados, 1113
clips de audio, 930
`Clock`, programa de ejemplo del directorio `applets`, 74
`close`, método, 771, 810, 836
  `FileChannel`, 814
`closePath`, método de `GeneralPath`, 537
COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios), 9, 22
código
  de operación, 294, 987
  de tecla virtual, 586
  en línea, poner el, 407
  fuente, 72
  nativo, 330
códigos de bytes, 13
coerción de argumentos, 197
cola
  de impresión, 967
  estructura de datos dinámica, 967
  tipo de datos abstracto, 345
colaboraciones (interacciones) entre los objetos, 277
colección, 1039
colecciones, 998
colisión
  de nombres, 339
  en un arreglo, 1005
colocar, operación relativa, 810
`Color`, clase, 510, 513
`Color.BLACK`, 631
`Color.GREEN`, 535
`Color.RED`, 535
`Color.WHITE`, 535
`Color.YELLOW`, 536
columna
  autoincrementada, 1085
  nombre, 1093
  número, 1093
columnas, 1075
  de tablas, 270
`Collection`
  clase, 1001
  interfaz, 1039
Comando, patrón de diseño, 667
comandos, 984
comas para separar argumentos en un método, 38
comentario, 28
  de fin de línea, 28
  de fin de línea, estilo C++, 29
  de múltiples líneas, 29
  estilo Java, 29
  Javadoc, 29
  tradicionales, 29
comentarios
  de fin de línea de Java (`//`), 1155
  de JSP, 1155
  de lenguaje de secuencia de comandos, 1155
  de XHTML, 1155
  tradicionales (delimitados por `/*` y `*/`), 1155
Comparable, interfaz, 1050
comparación lexicográfica, 458, 1052
`compareTo`, método de `String`, 457
compartido, bloqueo, 811
compilador, 486
compilador JIT ("just-in-time", justo a tiempo, 15
compiladores, 22
complemento a uno, 1024
Complemento de Java (Java Plug-in), 1159
complemento de Java, 85
Complemento Java (Java Plug-in), 72
`Component`, clase, 512, 550, 612, 923
componente
  de escucha del evento, 555
  de interfaz gráfica de usuario (GUI), 192
  de la GUI ligero, 624
  de origen, 634
componentes
  arreglos, 246
  de escucha de eventos, 498
  de la GUI, 38, 548
  de la GUI de Swing, 549
  de Swing, 549
  del sistema, 58
  ligeros, 550
  pesados, 550
  puros de Java, 549
  reutilizables de software, 22
  reutilizables, 18
  Swing de Java, 440
  UML, 59
comportamientos, 302
  de los objetos, 17
  de un objeto, 80
Compuesto, patrón de diseño, 665
computación
  cliente/servidor, 21
  crítica para los negocios, 683
  de misión crítica, 683
  distribuida, 21
  personal, 21
computadoras cliente, 21
comunicaciones
  basadas en flujos, 826
  basadas en paquetes, 826
  basadas en sockets, 827
`concat`, método de `String`, 463
concatenación de cadenas, 44
concurrencia del conjunto de resultados, 1097
condición
  de continuación de ciclo, 148
  de continuación del ciclo, 107
  de guardia (condiciones de seguridad), 108
  de guardia, 178
  de la repetición, 146
condiciones simples, 167
ConductoElevador, 133
`ConductoElevador`, clase del elevador, 95, 277, 436, 539, 739, 934
configuración
  objeto implícito, 1154
  regional, 155
conflicto de nombres, 339
conjunto
  codificado de caracteres, 811
  de bits, 1028
  de caracteres Unicode, 453
  de caracteres, 68, 751
ConnectException, clase, 837
Consorcio World Wide Web (W3C), 832

http://librosysolucionario.net

constante, 251, 987
con nombre, 252
de tecla, 586
constantes, 208
con nombre, 208
constructor, 313
de una clase, 208
predeterminado, 313
sin argumentos, 314
constructores sobrecargados, 313
consultas, 1074
SELECT, 1080
consumidor, 709
contador, 114
de ciclo, 146
de datos, 989
de instrucciones, 989
Container, clase, 192, 510, 550
contains, método de Vector, 1008
contenedor
de applets, 72
de JSPs, 1151
de servlets, 1114
contentType, atributo de la directiva page, 1172
conteo de base cero, 149
contexto de gráficos, 511
continue, palabra reservada, 162, 686, 876
contraseña, 556
control del programa, 105
controladas por eventos, las GUIs, 554
controlador, 656
de JDBC, 1074
controles, componentes de GUI, 548
conversión
descendente, 394
explícita, 124
implícita, 124
coordenada x (la coordenada horizontal), 510
coordenada y (la coordenada vertical), 510
coordenadas x-y, 82
copia del argumento, 258
copy, método de Collections, 1050
countTokens, método de StringTokenizer, 482
Courier, fuente 270
Craps, applet, 417
crear una instancia, 123
createGlue, método estático de Box, 647
createGraphics, método de BufferedImage, 535
createHorizontalBox, método de Box, 646
createHorizontalGlue, método de Box, 647
createHorizontalStrut, método de Box, 647
createImage, método de Component, 664
createRigidArea, método de Box, 647
createVerticalBox, método de Box, 646
createVerticalGlue, método de Box, 647
createVerticalStrut, método de Box, 647
criterios de selección, 1081
cuadro
combinado, 548
de selección o lista desplegable, 77
de desplazamiento, 569
de diálogo de entrada, 40
de diálogo modal, 518, 630
delimitador, 102, 137
cuadros de diálogo, 36
cuantificadores, 490
cuanto de tiempo, 701
cuenta de caracteres, 454
cuerpo
de la declaración de una clase, 31, 305
del método, 194
currentThread, método de Thread, 737
currentTimeMillis, método de la clase System, 225
cursor, 1092
de salida, 32
del ratón, 39
ChangeEvent, clase, 624
ChangeListener, interfaz, 624
Channel, clase, 810
Character
clase, 433
tipos, 1170
CharArrayReader, clase, 755
CharArrayWriter, clase 755
charAt, método de String, 455
CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, clases, 810
Charset, clase, 812
checkForTriggerEvent, método utilitario privado, 634

D

DatagramPacket, clase, 849
DatagramSocket, clase, 849
DataInput, interfaz, 754
DataInputStream, clase, 753, 754
DataOutputStream, clase, 753, 754
Date, clase, 1160
DateFormat, clase, 1160
datos
de entrada, 14
de plantilla fija, 1151
de salida, 14
de un objeto, 80
persistentes, 750
procesamiento, 21
DecimalFormat, 123
clase, 306, 313
declaración
de clase, 30
de un método, 32
import, 37
de variables, 147
de un método, 194
decremento, cantidad de, 146
definición
de un método, 32
de una clase, 30
degradado, 534
DELETE, instrucción, 1086
delete
método de StringBuffer, 473
petición de HTTP, 1116
deleteCharAt, método de StringBuffer, 473
demo, directorio del J2SDK, 73
dequeue, operación de una cola, 967
desaprobados, métodos, 733
desarrollo
de programas, 28
rápido de aplicaciones (RAD), 343
desbordamiento aritmético, 344
descartarlo (ocultarlo), 39
description, elemento de XML, 1125
descriptor de despliegue, 1124
deserialización de datos, 764
despachar
el subproceso, 702
evento, 560
desplazamiento
a la derecha con signo (>>), 1015
a la derecha sin signo (>>>), 1015
a la izquierda (<<), 1015
de caracteres en un arreglo, 454
valor de, 200
desplazar rango de números, 200
desreferenciados, objetos, 701
destroy
método de servlet, 1151
palabra clave, 214
devedge.netscape.com/library/manuals/2000/javascript/1.3/reference/cookies.htm, 1143
devolver resultados, 189
diagrama
con elementos omitidos (elided), 96
de actividad, 106, 177
de caso-uso de UML, 596
de clases, 96
de colaboración, 278
de interacción, 278
de secuencia, 278, 739
diagramas
de actividad, 59
de caso-uso, 59
de clases, 58
de colaboración, 59
de componentes, 59
de distribución, 59
de estado, 177
de estados, 59
de objetos, 58
de secuencia, 59
de transición de estado, 177
digit, método estático de Character, 477
dígitos decimales, 751
Dimension, clase, 617
dir, comando de MS-DOS, 73
dirección IP del servidor, 837
directiva include, 1159
directivas, acciones, scriptlets, 1150
DIRECTORIES_ONLY, constante, 765
directorio raíz, 756
discos, 14
diseño de un proyecto, 18
Diseño orientado a objetos (DOO), 17, 133
display-name, elemento de XML, 1125
dispose, método de Window, 624
DISPOSE_ON_CLOSE, constante de JFrame, 433
DO_NOTHING_ON_CLOSE y HIDE_ON_CLOSE, 624
dispositivos
de almacenamiento secundario, 750
de entrada, 21
de salida, 21
distribución de los componentes de GUI, 74
DitherTest, programa de ejemplo del directorio applets, 74
Dive Into™, 33
división de enteros, 46, 124
DO_NOTHING_ON_CLOSE, constante de JFrame, 433
documentación del API de Java, 37
documentar programas, 28
documento
de requerimientos del sistema, 303
en HTML, 13
doDelete, método de HttpServlet, 1116
doHead, método de HttpServlet, 1116
doOptions, método de HttpServlet, 1117
doPost, método de HttpServlet, 1116
doPut, método de HttpServlet, 1117
doTrace, método de HttpServlet, 1117
Double
clase, 433, 999
de java.lang, 1170
palabra reservada, 836, 1170
tipo de datos, 124, 532
tipo primitivo, 87, 92

Double.parseDouble, método static de Double, 195
Double.parseDouble, método static de Double, 92
downcasting (conversión descendente), 394
draw, método, 535
draw3DRect, método de Graphics, 526
drawArc, método de Graphics, 527
drawLine
método, 87
método de Graphics, 449
drawOval, método de Graphics, 102, 157, 527
drawPolygon, método de Graphics, 529, 530
drawRect, método de Graphics, 92, 306, 515
drawRoundRect, método de Graphics, 526
drawString, drawLine, drawOval, métodos de Graphics, 306
drawString, método de Graphics, 515, 82
DrawTest, programa de ejemplo del directorio applets, 74

E

E/S sin bloqueo, 897
EAST, constante de GridBagConstraints, 649
efecto secundario, 169
EfectosDeSonido, clase del elevador, 934
eje x, 510
eje y, 510
ejecución secuencial, 105
elemento
applet de HTML, 83
cero, 246
de azar, 199
de menú, 625
meta, 1153
elementos
de secuencias de comandos, 1151
de un arreglo, 246
del arreglo, 1043
Elevador, 133
clase, 95, 277, 434, 538, 738, 934
elevadorLlego, mensaje del elevador, 277, 739
elevadorPartio, mensaje del elevador, 277
eliminación de valores duplicados, 976
eliminación del goto, 105
Ellipse2D.Double, 532
Ellipse2D.Float, 532
emacs, editor de texto, 12
Empleado, clase, 326, 391
EmpleadoAsalariado, clase, 391
EmpleadoBaseMasComision, clase, 391
EmpleadoPorComision, clase, 391
EmpleadoPorHoras, clase, 391
empujar el plato en la pila, 344
En ejecución, estado de un subproceso, 702
En espera, estado de un subproceso, 703
encabezado
de flujo, 842
de la instrucción for, 149
del for, 149
del método, 194
encapsulamiento, 302, 362
encapsular, 302
encenderLuz
mensaje del elevador, 277
operación del elevador, 228
endsWith, método de String, 459
enlace, clase, 345
enfocado, campo, 556
enlace, 953
enlazar el servidor al puerto, 835
enqueue, operación de una cola, 967
EnteroEnorme, clase, 352
entidad externa, 597
Entorno Java 2 en tiempo de ejecución (J2RE), 100
entornos integrados de desarrollo (IDEs), 13, 23
entrarElevador
mensaje del elevador, 277
operación del elevador, 228
Enumeration
getParameterNames(), 1117
Enumeration
interfaz, 1002
objeto, 1043
enviar un mensaje, 81
a un objeto, 147
envoltura
de objetos flujo, 760
de tipos, 92
envolturas no modificables, 1066
equals
método de Arrays, 1039
método de String, 353, 457
equalsIgnoreCase, método de String, 457
equipo remoto, 13
error
de sintaxis, 29
en tiempo de petición, 1151
lógico fatal, 112
lógico no fatal, 112
lógico, 43, 112
por desplazamiento en uno, 148
Error, clase, 684
errores
de "desplazamiento en uno", 247
de compilación, 29
de truncamiento, 198
del compilador, 29
en tiempo de compilación, 15
en tiempo de ejecución, 14
en tiempo de traducción, 1151
fatales en tiempo de ejecución, 14
fatales, 299
no fatales en tiempo de ejecución, 14
sincrónicos, 677
errorPage, atributo de la directiva page, 1172
escalar rango de valores, 200
escrituras esparcidas, 811
EscuchaBoton, clase del elevador, 539
EscuchaLuz, clase del elevador, 539
EscuchaMoverElevador, clase del elevador, 539
EscuchaPuerta, clase del elevador, 538
escuchar eventos, 209
EscuchaTimbre, clase del elevador, 539
espacio
en blanco, 30, 486
libre horizontal, 592
libre vertical, 592
especificación del lenguaje Java, 30
esqueleto del simulador, 346
esquema
de catalogación, 343
de codificación de caracteres, 811
de un subprograma, 193
establecer, método, 319
establecerMovimiento, mensaje del elevador, 740
establecerUbicacion, mensaje del elevador, 740
estaciones de trabajo, 6
estado
de acción, 106
de un objeto, 59, 177
final, 106
inicial, 106
Listo de un subproceso, 702
o condición, de un objeto, 133
Estado, patrón de diseño, 668
estados de subproceso, 702
estereotipos, 538
Estrategia, patrón de diseño, 668
estructura
del sistema, 58
de repetición, 106
de secuencia, 106
de selección doble, 107
de selección múltiple, 107
de selección simple, 107
de selección, 106
estructuras
de ciclo, 107
de control, 105
de datos, 246, 952
dinámicas, 952
lineales, 970
de una sola entrada/una sola salida, 108
etiqueta, 164
de la casilla de verificación, 564
del botón, 561
etiquetas, 551
de HTML, 83
evaluación de corto circuito, 169
EventListenerList, clase, 560
evento, 207, 512
de cierre de ventana, 432
de desencadenamiento del menú contextual, 632
mouseMoved, 927
eventos, 177, 278
de ratón, 575
de ratón, eventos de tecla, 560
de teclas, 584
de ventana, 625
EventType, clase, 835
EventType.ACTIVATED, 835
EventType.ENTERED, 835
EventType.EXITED, 835
excepción
ante un índice inválido, 257
ClassCastException, 1167
no atrapada, 678
NoSuchElementException, 1001
NullPointerException, 1008
excepciones
encadenadas, 692
no verificadas, 684
verificadas, 684
Exception, clase, 684
exception, objeto implícito, 1154
exclusión mutua, 708
excluyente, bloqueo, 811
exists, método de File, 759
exit
método de System, 732
método estático de la clase System, 39
EXIT_ON_CLOSE, 624
expresión, 195
condicional, 110
controladora, 160
de acceso a un arreglo, 246
de acción, 106
de creación de arreglos, 271
de creación de instancia de una clase, 123
integral constante, 161
para crear un arreglo, 248
expresiones
de tipo mixto, 197
partes de una instrucción, 44
regulares, 486

http://librosysolucionarios.net

extends
  atributo de la directiva page, 1172
  palabra reservada, 79, 305, 366
extensión de archivo, 12

F

fábrica abstracta, 908
fabricantes independientes de software (ISVs), 385
factor
  de carga, 1005
  de escala, 200
factorial, 218
factoriales, 183
falsa, condición, 49
false, palabra reservada, 170, 309, 612, 740
fase
  de ejecución, 23
  de inicialización, 119
  de procesamiento, 119
  de recopilación de requerimientos, 596
  de terminación, 119
Fast CGI, 1115
Fecha, clase, 326
Figura, clase, 357
filas
  de una tabla, 270
  SQL, 1075
File, clase, 755
File.pathSeparator, 760
FileChannel, clase, 811
FileOutputStream, clase, 753
FileReader, clase, 753
FILES_AND_LIRECTORIES, constante, 765
FILES_ONLY, constante, 765
FileWriter, clase, 753
FilterInputStream, clase, 753
FilterOutputStream, clase, 753
fill
  campo de GridBagConstraints, 648
  método de Arrays, 1039
  método de Graphics2D, 534
fill3DRect, método de Graphics, 526
fillArc, método de Graphics, 527
fillOval, método de Graphics, 527, 613
fillPolygon, método de Graphics, 530
fillRect, método de Graphics, 515, 613
fillRoundRect, método de Graphics, 526
final, palabra reservada, 207, 251, 334, 407
finalize
  método, 381
  palabra reservada, 329
finally
  cláusula, 678
  palabra reservada, 842
firma
  de un constructor, 314
  de un método, 215
first, método, 1063
flecha derecha, tecla, 620
flecha hacia abajo, tecla, 621
flecha hacia arriba, tecla, 621
flecha izquierda, tecla, 620
flechas de desplazamiento, 569
flechas de transición, 106
flip, operación volteur, 812
float y double, tipos, 42
Float, 1170
  clase, 433
  tipo primitivo, 87
FlowLayout
  administrador de esquemas, 207, 644
  clase, 578, 590
FlowLayout.CENTER, 554, 590
FlowLayout.RIGHT, 554, 590
flujo, 752
  de control, 55
  de trabajo, 106, 177
  del control, 93
flujos continuos, 827
flush
  atributo, 1159
  método, 771
Font, clase, 270, 510
Font.BOLD, 270, 520, 565
Font.BOLD + Font.ITALIC, 270
Font.ITALIC, 270, 520, 566
Font.PLAIN, 270, 520, 565
FontMetrics, clase, 510, 523
for, palabra reservada, 107, 148, 631
forDigit, método de Character, 477
Forma, clase abstracta, 390
forma cíclica (round-robin), 701
format
  del objeto DateFormat, 1162
  método de DecimalFormat, 307
formato
  de hora universal, 305
  de línea recta, 47
  de reloj de 24 horas, 305
formulario, 1116
forName, método estático de Class, 417
FORTRAN (FORmula TRANslator), 9, 22
Fractal, programa de ejemplo del directorio applets, 74
Frame, clase, 584
frozen, término, 435
fuga
  de memoria, 701
  de recursos, 329, 678
función, 303
  de C, 17
funciones, 189
Fundación de software Apache (www.apache.org), 1113

G

g
  contexto gráfico, 81
  objeto grafica, 306
g.drawLine, método de Graphics, 195
GatheringByteChannel, clase, 811
gc
  método estático de la clase System, 385
  método static, 333
generalización de clases, 435
GeneralPath, clase, 510, 536
GenericServlet, clase, 1115
get
  método de HashMap, 828
  método de petición, 1116
getAbsolutePath, método de File, 760
getActionCommand, método de ActionEvent, 482, 559, 835
getBlue, método de Color, 514
getByName, método estático de InetAddress, 848
getClass, método de Object, 417
getClassName
  método de StackTraceElement, 692
  método de la clase UIManager.LookAndFeelInfo, 638
getClickCount, método de MouseEvent, 584
getCodeBase, método de Applet, 930
getColor, método, 514
getCurrencyInstance, método de NumberFormat, 155
getChannel, método de File, 811
getChar, método de ByteBuffer, 814
getChars, método de String, 455
getDocumentBase, método de Applet, 922, 930
getDouble, método de ByteBuffer, 814
getEventType, método de HyperlinkEvent, 835
getFamily, método de Font, 519, 521
getFileName, método de StackTraceElement, 692
getFont, método de Graphics, 519
getFontMetrics, método de Graphics, 523
getGreen, método de Color, 514
getHeight, método de Component, 923
getIcon, método de JLabel, 554
getImage, método, 922
getInsets, método, 510
getInstalledLookAndFeels, método de UIManager, 638
getInt, método de ByteBuffer, 814
getKeyCode, método de KeyEvent, 586
getKeyModifiersText, método de KeyEvent, 586
getKeyText, método de KeyEvent, 586
getLength, método de DatagramPacket, 852
getLineNumber, método de StackTraceElement, 692
getMessage, método de Throwable, 690
getMethodName, método de StackTraceElement, 692
getMinimumSize, método de Component, 621, 926
getModifiers, método de InputEvent, 586
getName
  método de Class, 417
  método de Font, 519, 521
  método de Thread, 707
getOutputStream, método de Socket, 836
getParameter, método de Applet, 828
getPassword, método de JPasswordField, 559
getPath, método de File, 760
getPoint, método de MouseEvent, 581
getPort, método de DatagramPacket, 852
getPreferredSize, método de Component, 617, 926
getPriority, método de Thread, 704
getProperty, método de Properties, 1009
getRed, método de Color, 514
getSelectedFile, método, 770
getSelectedIndex, método de JComboBox, 571
getSelectedText, método de JTextComponent, 611
getSelectedValues, método de JList, 575
getSize, método de Font, 519, 521
getSource(), método de ActionEvent, 323, 559, 631
getStackTrace, método de Throwable, 690
getStateChange, método de ItemEvent, 571
getStyle, método de Font, 519, 521
getText, método de JLabel, 554
getURL, método de HyperlinkEvent, 835
getValue, método de JSlider, 624
getWidth, método de Component, 923
getX, método de MouseEvent, 579
getY, método de MouseEvent, 579
GIF (Formato de intercambio de gráficos), 553, 922
goto, instrucción, 105
GradientPaint, clase, 510
grados
  negativos, 527
  positivos, 527
Gráficos de tortuga, 286, 545
Graphics g, 512
Graphics, clase, 79, 306, 449, 510, 511, 613
Graphics2D, clase, 532
Graphics-Environment, 270
GraphicsTest, programa de ejemplo del directorio applets, 74
GraphLayout, programa de ejemplo del directorio applets, 74
GridBagConstraints, clase, 647
GridBagConstraints.BOTH, 649
GridBagConstraints.CENTER, 615, 649
gridheight, campo de GridBagConstraints, 648
GridLayout, administrador de esquemas, 615
gridwidth, campo de GridBagConstraints, 648
gridx, campo de GridBagConstraints, 648
gridy, campo de GridBagConstraints, 648
grupo de transmisión múltiple, 878
Guardar, cuadro de diálogo de JFileChooser, 770
GUI (interfaz gráfica de usuario), 658
  ligera de Swing, 612
guión bajo (_), 1081

H

hablar de objetos, 302
hacer un ciclo, 117
hardware, 2
hashing, 998, 1005
HashMap, clase, 828, 1063
HashSet, clase, 1060
Hashtable, clase, 998
hasMoreElements, método de Enumeration, 1002
hasMoreTokens, método de StringTokenizer, 482
head, petición de HTTP, 1116
height, altura de un subprograma, 83
herencia, 79, 303, 356
  de implementación, 400
  estructural, 359
  múltiple, 17, 356
  simple, 356
Herencia, programación orientada a objetos, 355
hermanos, nodos, 970
heurística de accesibilidad, 289
heurística, 289
HIDE_ON_CLOSE, constante de JFrame, 433
hijo, nodo, 970
hipervínculos, 835
histograma, 184, 253
Historia de C++, 7
hoja, nodo, 665
.htm, 83
.html, 83
HTML (Lenguaje de marcado de hipertexto), 13, 83
HTTP, tipos de peticiones, 1116
HttpServlet, clase, 1115
HttpServletRequest, clase, 1116
HttpServletResponse, clase, 1154
HyperlinkEvent, clase, 832
HyperlinkListener, clase, 835, 912

I

Icon, clase, 553
ID de evento, 560
id, atributo, 1167
identificador, nombre de la clase, 30
Identificador uniforme de recursos (URI), 756
i-ésimo elemento del arreglo, 246
if, else, switch, while, do, palabras reservadas, 107
if, palabra reservada, 107, 631
if...else, instrucción, 107, 432
igualar las firmas, 314
ij, Cloudscape, 1087
IllegalArgumentException, clase, 842
IllegalThreadStateException, clase, 732
Image, clase, 921
ImageIcon, clase, 553, 921
ImageMap, programa de ejemplo del directorio applets, 74
ImageObserver, interfaz, 923
implements ActionListener, instrucción, 417
implements, palabra clave, 417
import, atributo de la directiva page, 1172
impresoras, 14
Inactivo, estado de un subproceso, 703
include, directiva, 1171
incremento
  de capacidad, 998, 1001
  cantidad de, 146
  de bloque, 621
indexOf, método de String, 461
indicador, 620
  mensaje, 43
índice, 569
  cero, 246
InetAddress, clase, 842
info, atributo de la directiva page, 1172
información sobre la ruta, 756
ingeniería
  hacia delante, 345
  inversa, 345
inicialización de variables, 147
inicializador de arreglo, 250
inicializadores, 313
  de arreglos anidados, 271
init
  método, 214
  método de JApplet, 81, 191, 512
  método de servlet, 1151
init-param, elemento, 1140
InputEvent, clase, 576, 584
InputStream, clase, 753, 836
INSERT, instrucción, 1085
Insets, clase, 510
instanceof
  método, 1046
  palabra reservada, 415
instancia, 80
  de clase, 80
instanciar, 80
  muchos objetos, 18
  objetos, 303
  un objeto, 123
Instituto nacional estadounidense de estándares (ANSI), 7
instrucción, 32, 984
  asistida por computadora (CAI), 240
  break, 161
    etiquetada, 164
  continue etiquetada, 166
  de control de una sola entrada/una sola salida, 172
  de declaración de variable, 90
  de un método, 194
  do...while, 156
  if
    condición, 49
    decisión en una, 49
    nula, 113
  sincronizada, 708
  switch de selección múltiple, 158
  vacía, 54, 113
instrucciones de control
  ejecutables, 105
  if...else anidadas, 110
  para declarar variables; 42
  Parte 1, 103
  try, 678
int
  palabra reservada, 307, 836
  tipo de datos, 42
Integer
  clase, 210, 433, 680, 993, 999
  tipo de datos, 117
  tipos, 1170
Integer.parseInt, método, 195
interbloqueo, 709
interfaces, 17, 207, 303
  de las colecciones, 1039
  gráficas de usuario, 38
  public, 306
interfaz, 534
  ActionListener, 417
  de marcado, 764
  de múltiples documentos (MDI), 608
  de Programación de Aplicaciones de Java, 188
  de seguimiento SingleThreadModel, 1116
  de usuario, 1136
  Enumeration, 998
  gráfica de usuario (GUI), 548
  nativa de Java (Java Native Interface), 1091
  pública de la clase, 310
  Servlet, 1115
  SwingConstants, 553
interlineado, 521
Internet, 22
Internet Explorer, 1120
  botones, 38
  intérprete de Java, 13
interrupt, método de Thread, 703
InterruptedException, clase, 703
intervalo de actualización, 1153
Invocación de métodos remotos (RMI), 1112
  sobre el Protocolo Inter-Orb de Internet (RMI-IIOP), 1113
invocar un método, 81, 189
invokeLater, método estático de SwingUtilities, 723
IOException, clase, 684, 760, 835, 1116
IsAbsolute, método de File, 760
isActionKey, método de KeyEvent, 586
isAltDown, isControlDown, isMetaDown e isShiftDown, métodos de KeyEvent, 586
isBold, método de Font, 519
isDaemon, método, 732
isDefined, método de Character, 475
isDigit, método de Character, 477
isDirectory, método de File, 759
isEmpty, método de Map, 1066
isErrorPage, atributo de la directiva page, 1172
isFile, método de File, 756
isItalic, método de Font, 519

http://librosysolucionarios.net

isJavaIdentifierPart, método de Character, 477
isJavaIdentifierStart, método de Character, 477
isLetter, método de Character, 477
isLetterOrDigit, método de Character, 477
isLowerCase, método de Character, 477
isMetaDown, método de KeyEvent, 583
isOpen, método de Channel, 810
isPlain, isBold e isItalic, métodos de Font, 521
isPlain, método de Font, 519
isPopupTrigger, método de MouseEvent, 634
isSelected
  método de AbstractButton, 631
  método de JCheckBox, 565
isThreadSafe, atributo de la directiva page, 1172
isUpperCase, método de Character, 477
ItemEvent, clase, 565
ItemListener, clase, 565
itemStateChanged, método de ItemListener, 565
iterador bidireccional, 1048
Iterator, objeto, 1043
IUBanco, clase, 761

J

J2SE 1.4, 750
.jar, extensión de archivo, 340
.java, extensión, 304
jakarta.apache.org/tomcat/index.html, página Web, 1143
JApplet, clase, 79, 192, 427, 512, 550, 612, 923
Java, 152, 921
Java Foundation Classes, Clases Fundamentales de Java, 73
Java Sound API, 921
java.apache.org, paquete, 1143
java.applet, paquete, 79, 199
java.applet.Applet, paquete, 550
java.awt, paquete, 79, 199, 532, 921
java.awt.Applet, paquete, 587
java.awt.Button, clase, 666
java.awt.Color, clase, 516
java.awt.color, paquete, 532
java.awt.Component
  clase, 428, 617
  paquete, 575
java.awt.Container, clase, 665, 668
java.awt.event, paquete, 90, 199, 432, 554, 579
java.awt.font, paquete, 532
java.awt.Frame, clase, 618, 624
java.awt.geom, paquete, 532
java.awt.Graphics, clase, 89
java.awt.image, paquete, 532
java.awt.image.renderable, paquete, 532
java.awt.Panel, paquete, 587
java.awt.print, paquete, 532
java.awt.Window, clase, 624
java.awt.Window, paquete, 550
java.io, 744
  paquete, 199, 753
java.io.IOException, clase, 1120
java.lang, paquete, 117, 199, 305, 356, 999
java.lang.Cloneable, interfaz, 1067
java.lang.Object, clase, 1154
java.lang.SingleThreadModel, clase, 1172
java.lang.Throwable, clase, 1154
java.math, paquete, 220
java.math.BigDecimal, clase, 156
java.net, 744
  paquete, 199, 826, 1112
java.nio.charset, paquete, 812
java.text, paquete, 123, 155, 199, 306, 1160
java.util, paquete, 155, 199, 828, 998, 1153
java.util.Observable, clase, 667
java.util.Observer, clase, 667
Java2D, subprograma, 77
JavaBeans, 1150
javac, compilador, 338
JavaServer, 1150
javax.swing.JmenuItem, clase, 625
javax.servlet, paquete, 1113
javax.servlet.http, paquete, 1113, 1154
javax.servlet.http.HttpSession, clase, 1154
javax.servlet.jsp, paquete, 1113
javax.servlet.jsp.JspWriter, clase, 1154
javax.servlet.jsp.PageContext, clase, 1154
javax.servlet.jsp.tagext, paquete, 1113
javax.servlet.ServletConfig, clase, 1154
javax.servlet.ServletContext, clase, 1154
javax.servlet.ServletException, clase, 1120
javax.swing, paquete, 79, 155, 199, 427, 513, 765, 921
javax.swing.AbstractButton, clase, 625
javax.swing.event, paquete, 199, 554, 624
javax.swing.JApplet, clase, 198
javax.swing.JMenuItem, clase, n25
javax.swing.Jpanel, clase, 587
Jbutton, clase, 313
JButton, objeto, 611 JScrollPane, objeto, 611
JcolorChooser, clase, 516
JComboBox, clase, 568
JComponent, clase, 513, 550, 568, 612
JCheckBox, clase, 737
JDBC™, 750
JDesktopPane, clase, 638
JEditorPane, clase, 832
jerarquía
  de clases, 356
  de herencia, 357
  Punto3/Circulo4/Cilindro, 380
jerarquía de clases punto/círculo, 391
jerarquía de datos, 751
jfc, directorio del J2SDK, 73
JFileChooser, clase, 765
JFrame, clase, 427, 550, 612
JFrame.EXIT_ON_CLOSE, constante, 433
JinternalFrame, clase, 638
JLabel, clase, 207, 551, 737, 831
JList, clase, 571, 828
JMenu, clase, 625
JMenuBar, JMenu, JMenuItem, JcheckBoxMenuItem, clases, 625
JMenuItem, clase, 625
JoptionPane, clase, 156, 308, 361, 417
JOptionPane.ERROR_MESSAGE, 45
JOptionPane.INFORMATION_MESSAGE, 45
JOptionPane.PLAIN_MESSAGE, 45
JOptionPane.QUESTION_MESSAGE, 45
JOptionPane.WARNING_MESSAGE, 45
JPanel, clase, 595
JPanel, clase, 612
JPasswordField, clase, 556
JPEG (Grupo unido de expertos en fotografía), 553, 922
JPopupMenu, clase, 632
JRadioButton, clase, 563
JRadioButtonMenuItem, clase, 625
JScrollPane, clase, 155, 573
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS, 611
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED, 611
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER, 611
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS, 611
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED, 611
JScrollPane.VERTICAL_SCROLLBAR_NEVER, 611
jspDestroy, método, 1151
jspInit, método, 1151
jspinsider.com, 1186
_jspService, método, 1151
jsptags.com, 1186
JTabbedPane, clase, 642
JTextArea, clase, 155, 608, 760
JTextComponent, clase (paquete javax.swing.text), 556, 611, 842
JtextField, clase, 207, 297, 556, 680, 832
JTextField, objeto editable, 209
JToggleButton, JCheckBox, clases, 563
JumpingBox, programa de ejemplo del directorio applets, 74
justificar a la izquierda, 553
JVM, 13

K

KeyListener, clase, 560, 584
keyPressed, keyReleased, métodos de KeyListener, 584
keyTyped, método de KeyListener, 584
KIS, 14
Kit de desarrollo de software para Java 2 (J2SDK), 21, 760
kit de herramientas de ventana, 624

L

La criba de Eratóstenes, método para número primos, 290
La tortuga y la liebre, Simulación, 291
language, atributo de la directiva page, 1172
lanzar una excepción, 678
lanzar nuevamente, 685
last, método, 1063
lastIndexOf, método de String, 461
lastModified, método de File, 760
layoutContainer, método de LayoutManager, 590
LayoutManager
  clase, 587, 590
  interfaz, 668
LayoutManager2, clase, 590
lecturas esparcidas, 811
length
  campo de un arreglo, 247
  método de File, 760
  método de String, 455
lenguaje
  C, 7
  extensible, 307, 345
  fuertemente tipificado, 132
  natural, 6
  programación Ada, 10
Lenguaje Unificado de Modelado™ (UML), 16, 23, 106
lenguajes
  B, 7
  de alto nivel, 7, 21
  de computación, 21
  de programación por procedimientos, 303
  ensambladores, 6
  máquina, lenguajes ensambladores, 21
levantarse y arreglarse, algoritmo, 104
LIKE, operador, 1081
Line2D.Double, Rectangle2D.Double, RoundRectangle2D.Double Arc2D.Double, 532
línea
  de vida, 739
  para empezar a escuchar, 210
  punteada, 107
LineNumberReader, clase, 755
lineTo, método de GeneralPath, 537
LinkedList, clase, 1044
LinkedList, clase, 954
List
  interfaz, 1039
  clase, 1050
lista
  de enlace simple, 954
  de parámetros, 81
  desplegable, 568
  enlazada, 954
  separada por comas, 42
listas
  de selección múltiple, 571
  de selección simple, 571
  enlazadas, 952
ListIterator, objeto, 1044
ListSelectionEvent, clase, 571
ListSelectionListener, interfaz, 573
literal de cadena, 32
literales
  de cadena, 453
  de carácter, 453
load, instrucción, 991
Locale, clase, 1160
Locale.US, 155
localhost, 842, 1121
localización de la interfaz de usuario, 551
Localizador uniforme de recursos (URL), 756
lock, bloqueo de archivo, 811
lógica de negocios, 1136
lógico a nivel de bits (xor), 1028
Long
  clase, 433
  tipos, 1170
LookAndFeelInfo, clase anidada, 638
lookingAt, método de Matcher, 494
loop, método de AudioClip, 930
ls, comando de Unix, 73
Luz, clase del elevador, 95. 134, 177, 277
luzEncendida, atributo del elevador, 134
llamada
  a un método, 189
  de bloqueo, 876
  por referencia, 258
  por valor, 258
  recursiva, 218
  sincrónica, 279
llamado, atributo del elevador, 134, 179
llamar a un método, 189
llave izquierda, 31
llaves ({}), 678

M

main, método, 31, 115, 453, 705
MalformedURLException, clase, 831, 835
manejador, 310
  de eventos, 207
  de eventos, registro, 210
  de excepciones, 678
manejadores de eventos, 555
manejo
  de eventos, 207, 280
  de excepciones, 676
mantener
  el objeto en un estado consistente, 306
  la integridad del objeto, 306
Map, interfaz, 1002, 1039
mapas de imágenes, 927
MappedByteBuffer, clase, 810
Máquina virtual de Java, 13
marcado para la recolección de basura, 329
marcador de fin de archivo, 752
marcadores de visibilidad, 345
marcar instrucción, 988
marcas
  en un objeto JSlider, 620
  más distanciadas, 620
  menos distanciadas, 620
marco de trabajo de colecciones, 1038
marcos de destino, 832
  _blank, 832
  _top, 832
  de HTML, 832
  marco _self, 832
máscara, operando, 1016
matches, método, 486, 491
Math.min, 618
Math.PI y Math.E, constantes de Math, 190
Math.PI, 67
maximizar una ventana, 427
máximo común divisor (MCD), 239
mayúsculas y minúsculas, sensibilidad, 30
mecanismo de extensión, 340
  de marcas, 1151
mecanismo de manejo de excepciones, 638
media dorada, 220
mejoramiento de arriba a abajo, paso a paso, 118
Memento, patrón de diseño de comportamiento, 444
MemoryImageSource, clase, 948
mensaje drawLine, 147
mensajes, 177
menús, 38, 548
  contextuales sensibles al contexto, 632
metadatos, 1092
método
  absolute de ResultSet, 1099
  add de Vector, 1001
  addAll de List, 1048
  addTableModelListener, 1094
  arraycopy de System, 1041
  asList de Arrays, 1050
  clear de BitSet, 1028
  clear de ByteBuffer, 903
  clear de List, 1048
  clone de la clase java.lang.Object, 1067
  close de MulticastSocket, 887
  close de SocketChannel, 906
  compareTo, 1050
  configureBlocking, 898
  contains de Vector, 1001
  containsKey de Hashtable, 1008
  containsKey de Map, 1065
  createSocketImpl de la interfaz SocketImplFactory, 909
  createStatement de Connection, 1092
  de código activo, 2
  de comparación natural, 1050
  de hundimiento, 261
  decode de charSet, 903
  drawImage de Graphics, 921
  elements de Vector, 1002
  encode de Charset, 906
  encode de la clase Charset, 903
  equals de BitSet, 1029
  estático createHorizontalBox, 609
  estático open de la clase Selector, 902
  estático pow de la clase Math, 155
  executeQuery de Statement, 1177
  executeUpdate de Statement, 1177
  finalizador, 329
  finalize de Object, 1099
  fireTableStructureChanged (heredado de la clase AbstractTableModel), 1099
  firstElement de Vector, 1001
  flip, 903
  flush de la clase PrintWriter, 883
  get de BitSet, 1028
  get de Hashtable, 1008
  get de Map, 1065
  getAddress de DatagramPacket, 852
  getAudioClip de Applet, 930
  getClass (heredado de la clase Object), 893
  getCodeBase de JApplet, 862
  getColumnClass, 1094
  getColumnClassName de ResultSetMetaData, 1099
  getColumnCount, 1094
  getColumnName, 1094
  getColumnType de ResultSetMetaData, 1093
  getCookies() de Cookie, 1118
  getData de DatagramPacket, 884
  getHost de URL, 862
  getHostName de InetAddress, 842
  getInetAddress de Socket, 842
  getInitParameter de ServletConfig, 1140
  getLocale, 1160
  getObject de ResultSet, 1093
  getOutputStream, 1120
  getParameter, 1128
  getParameter, 1158
  getParameterValues del objeto implícito de JSP request, 1164
  getParameterValues, 1136
  getResource de la clase Class, 893
  getRow de ResultSet, 1099
  getRowCount, 1094
  getValueAt, 1094
  getWriter, 1120
  hasNext de Iterator, 1046
  headSet de TreeSet, 1062
  indexOf de Vector, 1001
  init, 1115
  insertElementAt, 1001
  instantiate de la clase java.beans.Beans, 1167
  invokeLater de SwingUtilities, 894
  isCellEditable, 1094
  isEmpty de Stack, 1004
  isEmpty de Vector, 1002
  isEmpty, 1009
  isOpen de SocketChannel, 903
  isReadable de la clase SelectionKey, 902
  iterator de Collection, 1046

http://librosysolucionarios.net

joinGroup de MulticastSocket, 884
keySet de HashMap, 1066
last de ResultSet, 1099
lastElement de Vector, 1001
leaveGroup de MulticastSocket, 887
list, 1014
listIterator de List, 1048
load de Properties, 1015
llamado (invocado), 189
manejador del evento, 555
next de Iterator, 1046
next de ResultSet, 1092
paintIcon, 921
parseInt, 993
play de Applet, 930
play de la interfaz AudioClip, 930
pop de Stack, 1004
predicado, 958
println de PrintWriter, 883
push de Stack, 1004
put de Hashtable, 1008
put de Map, 1065
que hizo la llamada, 189
random de la clase Math, 199
receive de DatagramSocket, 850
recursivo, 217
remove de Iterator, 1046
remove de Vector, 1002
removeAllElements, 1002
removeElementAt, 1002
removeTableModelListener de TableModel, 1094
repaint, 214
sendRedirect, 1134
set, 1001
set de BitSet, 1028
set de Iterator, 1048
setBorder de la clase JLabel, 894
setContentType, 1120
setEnabled, 893
setMnemonic, 893
setProperty de System, 1140
setSoTimeout de la clase Socket, 876
setValueAt, 1094
setWrapStyleWord, 893
size de BitSet, 1029
socket de la clase ServerSocketChannel, 897
sort de Collections, 1052
static getConnection de la clase DriverManager (paquete java.sql), 1092
static getDateTimeInstance de DateFormat, 1160
static open de la clase ServerSocketChannel, 897
stop de Timer, 926
store de Properties, 1014
subList de List, 1048
tailSet de TreeSet, 1063
toArray, 1049
toBinaryString, 1018
toString de BitSet, 1029
toString en el objeto CharBuffer, 903
toUpperCase de String, 1048
trimToSize de Vector, 999
utilitario privado, 634
write de la clase SocketChannel, 903
Método de fábrica, patrón de diseño, 664
Método de plantilla, patrón de diseño, 668
métodos, 8, 80, 303
  abstractos, 398
  ayudantes, 305
  de acceso, 306, 319
  de consulta, 319
  de envoltura, 1044
  de petición, 1116
  de vista de rango, 1061
  declarados por el programador, 189
  doGet, 1116
  init, start, paint, stop, 214
  manejadores de eventos, 209
  mutadores, 319
  predicado, 306
  sincronizados, 708
  utilitarios, 305
métrica de los tipos de letras, 521
MFC (Microsoft Foundation Classes), 22
Microsoft Foundation Classes, 10
Microsoft Internet Explorer, 13
Microsoft SQL Server, Oracle, Sybase, DB2, Informix, 1074
Microsoft Windows, 22
mid o .rmi (MIDI, Interfaz digital para instrumentos musicales), 930
miembros
  de datos, 303
  de una clase, 303
MiFigura, jerarquía, 449
MIME (Extensión multipropósito de correo Internet), 1120
minimizar una ventana, 427
MissingResourceException, mensaje de error, 84
modales, cuadros de diálogo, 630
modelo
  de delegación de eventos, 556
  de programación acción/decisión, 109
  de reanudación del manejo de excepciones, 679
  de software orientado a objetos, 296
  de terminación del manejo de excepciones, 679
ModeloElevador, clase del elevador, 436
modificadores de acceso, 305
modo
  de apertura del archivo, 787
  de selección, 572
módulo, operador de residuo, 46
módulos, 188
MoleculeViewer, programa de ejemplo del directorio applets, 74
moneyFormat.format, método, 155
monitores, 708
Monospaced
  SansSerif y Serif, tipos de letra, 520
  tipo de letra, 270
montante vertical, 647
Motif (una apariencia visual de UNIX), 608
motor
  de servlets, 1114
  de sonido, 930
MouseAdapter, clase adaptadora, 617
mouseClicked, método de MouseListener, 576
mouseDragged, método de MouseListener, 576, 618
mouseEntered, método de MouseListener, 576
MouseEvent, clase, 560
mouseExited, método de MouseListener, 576
MouseInputListener, clase, 575
MouseListener, clase, 560, 575, 634
MouseMotionAdapter, clase, 580, 617
MouseMotionListener, clase, 560, 575
mouseMoved, método de MouseListener, 618
mouseMoved, método de MouseMotionListener, 576
mousePressed, método de MouseListener, 575, 616, 748
mouseReleased, método de MouseListener, 576, 618
MouseWheelEvent, clase, 576
MouseWheelListener, interfaz, 576
mouseWheelMoved, método, 576
MoverElevador, mensaje del elevador, 278
moveTo, método de GeneralPath, 537
movimiento, atributo del elevador, 134
multimedia, 920
MULTIPLE_INTERVAL_SELECTION, 573
multiplicación, 46
multiplicidad, 96
Multiprocesadoras, 5
multiprogramación, 21
multitarea, 10, 22
mutuamente exclusivas, opciones, 566
MySQL, 1074

N

n!, 218
Nacimiento, estado de un subproceso, 702
name, atributo, 1171
navegabilidad, 346
navegador mínimo, 84
navegadores Web, 13
nemónicos, 551, 625
NervousText, programa de ejemplo del directorio applets, 74
Netscape Navigator, 13, 1120
new, palabra reservada, 123, 247, 306
newAudioClip, método estático de Applet, 944
nextElement, método de Enumeration, 1002
nextToken, método de StringTokenizer, 482
Niklaus Wirth, profesor, 10
nivel de información, 911
nivel de sangría, 31
nivel inferior, 911
no seleccionado, estado de JRadioButton, 566
nodos, 954
  hoja, 970
  raíz, 970
nombre
  completamente calificado, 339
  de archivo, 12
  de clase completamente calificado, 89
  de host, 1121
  de rol, 97
  de una variable, 42, 45
  de variable, 987
  del atributo, 135
  simple, 89
nombres de clase únicos, 336
NONE, VERTICAL, HORIZONTAL, constants de GridBagConstraints, 649
NORTH, constante de GridBagConstraints, 649
NORTHEAST, constante de GridBagConstraints, 649
NORTHWEST, constante de GridBagConstraints, 649
NOT lógico, 167
notación infijo, 980
notación postfijo, 980
notas de UML, 107
notify, método de Thread, 703
notifyAll, método de Thread, 703
Nueva API de E/S de Java, 486
Nuevas APIs de E/S, 750
null, palabra reservada, 38, 107, 385, 482, 581, 630
NullPointerException, clase, 754
Number, clase, 434
NumberFormat, clase, 155
NumberFormatException, clase, 680
número
  de línea, 984
  de puerto reconocido, 1122
  de puerto, 837
  perfecto, 239
  primo, 239
números
  de punto flotante, 87, 124
    con precisión doble, 87
    con precisión simple, 87
  seudoaleatorios, 199

O

Object Management Group, 19
Object, clase, 305, 356, 458, 496, 550, 683, 703, 753, 828, 999
ObjectInput, interfaz, 755
ObjectInputStream, clase, 753, 755
ObjectOutput, interfaz, 755
ObjectOutputStream, clase, 753, 755
objeto
  animados y objetos inanimados, 17
  colaboradores, 278
  contenedor, 306
  Cookie, 1118
  de contexto, 445, 668
  de estado, 445, 668
  de salida estándar, 32
  del evento, 555
  emisores, 279
  fachada, 909
  flujo de entrada estándar, 753
  flujo estándar de error, 753
  flujo estándar de salida, 753
  huérfanos, 701
  observadores, 667
  originador, 444
  pensar en, 302
  proxy, 443
  receptor, 279
  ServletRequest, 1116
  ServletResponse, 1116
  String anónimos, 453
  sujeto, 667
Observador, patrón de diseño, 667
Observer, interfaz, 667
obtener
  el bloqueo, 708
  métodos, 319
obtenerBoton, operación del elevador, 436
obtenerCapacidad, operación del elevador, 435
obtenerPuerta, operación del elevador, 436
  java.lang, método, 417
ocultamiento de información, 17, 302
Ocho reinas
  métodos de fuerza bruta, 290
  programa, 290
ODBC (Conectividad abierta de bases de datos), 1091
opaco, componente, 612
opción
  -classpath, 340
  de línea de comandos (-d), 338
opcionales, paquetes, 36
operación
  bloqueadora, 811
  física de entrada, 754
  física de salida, 754
  lógica de entrada, 754
operaciones, 96
  comportamientos, 133
  concurrentes, 700
  en los objetos, 17
  en paralelo, 700
  lógicas de salida, 754
  masivas, 1043
operador, 44
  "+", 487
  binario, 43
  condicional (?:), 110
  de asignación compuesto de suma, 129
  de asignación, 43
  de conversión de tipo, 68, 122, 197
  de decremento postfijo, 129
  de decremento prefijo, 129
  de expresión regular "*", 487
  de incremento postfijo, 129
  de incremento prefijo, 129
  de postincremento, 129
  de preincremento, 129
  residuo, 68
  ternario, 110
  unario, 124
  unario de conversión de tipo, 124
  unario de decremento, --, 129
  unario de incremento, ++, 129
operadores
  + y +=, 467
  aritméticos, 46
  de asignación a nivel de bits, 1024
  de asignación compuestos, 129
  de asignación compuestos aritméticos, 129
  de igualdad == y !=, 167
  de igualdad, 49
  lógicos, 167
  relacionales, 49
  relacionales >, <, >= y <=, 167
operando, 987
operandos, 43
oprimido, atributo del elevador, 134, 177
oprimirBoton
  mensaje del elevador, 739
  operación del elevador, 228
options, petición de HTTP, 1117
OR, 1028
  excluyente a nivel de bits (^), 1015
  excluyente lógico booleano, 167
  incluyente a nivel de bits (|), 1015
  incluyente lógico booleano, 167
orden de las acciones, 104
ordenamiento, 261
  burbuja, 261
  de árbol binario, 975
  estable, 1050
  por cubetas, 291
  por selección, 292
ORDER BY, cláusula, 1082
orientación
  a objetos, 16
  horizontal, 621
  vertical, 621
origen
  de confianza (trusted source), 869
  del evento, 323, 555
out, objeto implícito, 1154
OutputStream, clase, 753, 836

P

pack, método de JInternalFrame, 642
package, declaración, 337
page, objeto implícito, 1154, 1159, 1171
PageContext, clase, 1154
Pages (JSP), 1150
paint, método, 81, 214, 612
  de Applet, 923
  de Graphics, 306
  de JApplet, 148, 160, 512, 532
paintComponent, método, 612
  de JComponent, 513
palabras
  reservadas, 30
  Simpletron, 294
palíndromo, 143, 292
panel de contenido, 192
paneles, 395
PanelImagen, clase del elevador, 934
PanelMovil, clase del elevador, 934
Paquete Abstract Window Toolkit de Java, 199
Paquete Abstract Window Toolkit Event de Java, 199
Paquete Applet de Java, 199
Paquete de componentes GUI Swing de Java, 199
Paquete de entrada/salida de Java, 199
Paquete de red de Java, 199
Paquete de texto de Java, 199
Paquete de utilerías de Java, 199
Paquete del Lenguaje Java, 199
paquete java.awt, 192, 270, 549, 617
paquete java.awt.peer, 550
paquete java.io, 987
paquete java.nio.channels, 897
paquete javax.swing, 549
paquete javax.swing.event, 575
paquete predeterminado, 308
Paquete Swing Event de Java, 199
paquetes, 827, 1112
  comunicaciones basadas en, 1112
  de Java, 36
  en UML, 657
  java.rmi, 1112
  opcionales, 340
  org.omg, 1112
par de coordenadas, 82, 510
param
  atributo, 1171
  elemento, 828
parámetro, 193
  de excepción, 678
parámetros
  de applet, 827
  de inicialización, 1140
  de salida, 1104
  formales, 193
paréntesis redundantes, 49
pares clave/valor, 828
parseInt
  método de Integer, 680
  método estático, 43
partición de tiempo (timeslicing), 703
partidaElevador, operación del elevador, 228
pasadas de un compilador, 987
pasajero del elevador, 280
pasajeroEsperando, mensaje del elevador, 738
pasajeroViajando, mensaje del elevador, 739
Paseo del caballo, 287
  métodos de fuerza bruta, 289
  prueba del paseo cerrado, 290
paso
  de recursividad, 218
  por referencia, 258
  por valor, 258
patrón
  de arquitectura Niveles, 911
  de diseño
    Fábrica abstracta, 908
    Fachada, 909
    Iterador, 1068
  de URL, 1125
patrones

http://librosysolucionarios.net

de comportamiento, 440
de estructura, 440
de concurrencia, 743
patrones de diseño, 20, 440, 655
concurrentes, 440
de creación, 441
de diseño de estructura, 443
de Estado, 445
de estructura y patrones de diseño de comportamiento, 20
orientados a objetos, 20
Proxy, 443
PC (Computadora personal), 21
peek, método de **Stack**, 1004
pegamento horizontal, 647
PEPS (primero en entrar, primero en salir), 967
pequeños círculos, 106
Persona, clase del elevador, 95, 133, 177, 277, 434, 538, 656, 738, 934
personalización de la clase **JPanel**, 608
pila, 344, 963
de ejecución del programa, 963
PipedInputStream, clase, 753
PipedOutputStream, clase, 753
PipedReader, clase, 755
PipedWriter, clase, 755
Piso, clase del elevador, 95, 134, 434, 538, 656, 739
pisoActual, atributo del elevador, 134
pisoDestino, atributo del elevador, 134
píxeles, 23
PixelGrabber, clase, 948
planos de construcción, 80
plantillas, 80
plataforma .NET, 22
Plataforma Java 2, Standard Edition versión 1.4., 486
plataformas de hardware, 7
PNG (Gráficos portables de red), 533, 922
Point, clase, 581
polígonos, dibujo, 527
polilíneas, dibujo, 527
polimorfismo, 303, 386, 390
polinomio, 48
Polygon
clase, 510
paquete **java.awt**, 529
pop, operación de una pila, 963
por procedimientos, 17
porcentaje (%), 1081
portable, lenguaje, 14
position, método de **FileChannel**, 814
post, 1116
postdecremento, operador, 129
precedencia de operadores, 54
reglas, 47
predecremento, operador, 129
predicados, 1081
PreparedStatement, interfaz, 1104
primera mejora, 119
principio del menor privilegio, 334
print, instrucción, 35
printStackTrace, método de **Throwable**, 690
PrintStream, clase, 754
PrintWriter getWriter(), método, 1118
PrintWriter, clase, 755, 1120
prioridad, 701
private, palabra reservada, 305, 358, 427
probabilidad, 199
problema del **else** suelto, 111
procedimiento, 104
procedimientos, 189
almacenados, 1104
procesamiento, 836
avanzado de imágenes de Java, 921
proceso controlado por eventos, 512
Proceso de la comunidad Java (Java Community Process), 1113
Proceso de la comunidad Java (JCP, Java Community Process), 538
processActionEvent, método de **ActionListener**, 666
processEvent, método de **ActionListener**, 666
producciones, combinaciones de instrucciones, 993
productor, parte de una aplicación correspondiente al, 709
Programa de conversión al sistema métrico, 507
programa de edición, 12
programación, 303
basada en objetos (PBO), 303
controlada por eventos, 209
de subprocesos, 701
estructurada, 54, 93
orientada a objetos (POO), 3, 15, 356
preferente, 704
sin **goto**, 105
programadores de computadoras, 4
programas
de cómputo, 4, 21
intérpretes, 7, 22
tolerantes a fallos, 676
promoción
de argumentos, 197
operación de Java, 124
Properties, clase, 998
property, atributo, 1171
proporción dorada, número denominado, 220
protected, palabra reservada, 332, 359, 427
Protocolo de transferencia de hipertexto (HTTP), 827
protocolo HTTP (Protocolo de transferencia de hipertexto), 1114
Prototipo, 1067
proyecto Green, 8
Proyecto Jakarta, 1113, 1143
pseudocódigo, 18
public, palabra reservada, 30, 304, 305, 358, 427
publicar información, 827
Puente, patrón de diseño, 665
Puerta, clase del elevador, 436, 538
Puerta, Luz, Timbre, clases del elevador, 539
puertaAbierta, mensaje del elevador, 278
puertaCarrada, mensaje del elevador, 278
PuertaElevador, clase del elevador, 134, 277, 436, 95
puertaElevador, objeto, 739
PuertaPiso, clase del elevador, 95, 134, 277, 436
puntero del ratón, 39
punto
de entrada, 309
de inserción, 1041
de lanzamiento, 679
de negociación (handshake), 835, 1122
Punto, clase, 359, 390
Punto3, clase, 359, 392
push, operación de una pila, 963
put
método de **HashMap**, 828
petición de HTTP, 1117

Q

Quicksort, técnica de ordenamiento, 293
Quintillas, 304

R

raíz
(base) de un número, 477
de contexto, 1124
rama
flujo de actividad, 178
nodo, 665
RandomAccessFile, clase, 754, 776
rascio, cuantificador, 491
read
método de **FileChannel**, 814
método de **SocketChannel**, 836
Read, método, 811
ReadableByteChannel, clase, 811
readBoolean, **readByte**, **readChar**, **readDouble**, **readFloat**, **readFully**, métodos de **DataInput**, 754
readDouble, método, 784
Reader, clase, 755
readInt, método, 754, 784
readLine, método de **DataInput**, 754
readLong, método de **DataInput**, 754
readObject, método de **ObjectInput**, 755
readShort, método de **DataInput**, 754
readUnsignedByte, método de **DataInput**, 754
readUnsignedShort, método de **DataInput**, 754
readUTF, método de **DataInput**, 754
Thread, reanudarse el objeto, 737
recolección de basura, 329, 683, 701
recolector de basura, 329
recorrido
de laberintos, 293
en orden de niveles de un árbol binario, 976
recorridos inorden, preorden y postorden, 971
rectángulo
con relieve, 526
redondeado, 177
redes de área local, 6
redirigir flujos, 753
referencia
adelantada, 988
g de la clase **Graphics**, 147
this, 311
referencias, variables de tipo de referencia, 91
referirse a los objetos, 91
regionMatches, método de **String**, 457
registrarse para recibir un mensaje, 498
registro, 751
RegistroCuentas, clase, 761
regla
de anidamiento, 175
de apilamiento, 174
empírica, 289
Regla de integridad de entidades, 1078
Regla de la integridad referencial, 1077
reglas
básicas de alcance, 211
de promoción, 197
relación
cliente-servidor, 827, 1113
dos a uno, 96
"es un", 356
productor/consumidor, 709
"tiene un", 97, 356
uno a dos, 96
uno a uno, 96
uno a varios, 667
relacionar patrones, 1081
relaciones, 1078
relativas, operaciones, 810
repaint, método, 512, 612, 669
repetición
controlada por contador, 114
definida, 114
indefinida, 118
repite, el servidor, 849
replace, método de **String**, 464
replaceAll, método de **String**, 491
replaceFirst, método de **String**, 491
requerimiento
de atrapar o declarar, 684
del sistema, 596
recopilación, 58
request, objeto implícito, 1154
requestFocus, método, 556
response, objeto implícito, 1154
restablecerBoton, mensaje del elevador, 277
ResultSet, clase, 1177
ResultSet, interfaz, 1092
return, **break**, palabras reservadas, 686
return, palabra reservada, 193, 307
reutilización de software, 9, 189, 356
rombos, diagramas de actividad, 106
rotate, método de **Graphics2D**, 538
RoundRectangle2D, clase, 510
rt.jar, archivo de ficheros, 340
Runnable, interfaz, 723
ruta
absoluta, 756
de clases, 340
general, 536

S

sacar el plato de la pila, 344
Salir, elemento del menú **Subprograma** de **appletviewer**, 75
salirElevador, mensaje del elevador, 278
salirElevador, operación del elevador, 228
sangría, 31
SansSerif, tipo de letra, 270
ScatteringByteChannel, interfaz, 811
Scope, atributo, 1167
script de shell, 40
scriptlets, 1113
secuencia, 1044
de cuadro, 943
de escape, 36
de inicio, 82
de mensajes, 279
secuencias, 970
de escape, 1155
seek, método, 776
segunda mejora, 119
selección de disponibilidad, 897
SelectableChannel, interfaz, 897
SelectionKey.OP_ACCEPT, 902
SelectionKey.OP_READ, 902
selectores, 897
sembrar generador de números aleatorios, 200
separador
de directorio, 340
de menús, 631
de punto, 82
SequenceInputStream, clase, 755
Serializable, 836
interfaz, 755
serialización de objetos en Java, 1009
serializar objetos, 763
Serif, tipo de letra, 270, 631
ServerSocket, clase, 835
ServerSocketChannel, clase, 897
service, método de **HttpServlet**, 1115
servicio
orientado a la conexión, 827
sin conexión, 827
servicios **public**, 306
servidor, 827, 1113
de aplicaciones, 1137
Web, 13
servidores de archivos, 21
servlet, 1125
predeterminado, 1126
servlet.jsp.tagext, paquete, 1150
servlet-class, elemento de XML, 1125
ServletConfig, clase, 1140
ServletConfig
getServletConfig(), método, 1115
ServletException, clase, 1116
servlet-mapping, elemento de XML, 1125
servlet-name, elemento de XML, 1125
ServletOutputStream, clase, 1120
ServletOutputStream
getOutputStream(), método, 1118
servlets y JavaServer Pages (JSP), 1113
session, objeto implícito, 1154, 1172
Set, interfaz, 1039
setAlignment, método de **FlowLayout**, 590
setBackground, método de **Component**, 518, 573
setCaretPosition, método de **JTextComponent**, 842
setColor, método de **Graphics**, 514
setConstraints, método de **GridBagLayout**, 652
setDaemon, método de **Thread**, 732
setDefaultCloseOperation, método de **JFrame**, 624
setDisabledTextColor, método, 586
setEditable, operación del elevador, 209
setEnabled, método, 486
setErr, método de **System**, 753
setFixedCellHeight, método de **JList**, 575
setFixedCellWidth, método de **JList**, 575
setFont
método, 270
método de **Graphics**, 519
setHorizontalAlignment, método de **JLabel**, 553
setHorizontalScrollBarPolicy, método de **JScrollPane**, 611
setHorizontalTextPosition, método de **JLabel**, 554
setIcon, método, 553, 554
setIn, método de **System**, 753
setInverted, método de **JSlider**, 621
setJMenuBar, método, 625
setLayout
método de **Container**, 578, 668
operación del elevador, 208
setLength, método de **StringBuffer**, 469
setLineWrap, método, 611
setListData, método de **JList**, 575
setLocation, método de **Component**, 624
setLookAndFeel, método, 638
setMajorTickSpacing, método, 621
setMaximumRowCount, método de **JComboBox**, 569
setMnemonic, método de **AbstractButton**, 630
setOpaque, método de **JComponent**, 612
setOut, método de System, 753
setPage, método de **JEditorPane**, 835
setPaint, método de **Graphics2D**, 532
setPaintTicks, método de **JSlider**, 621
setPriority, método de **Thread**, 704
setProperty, método de **Properties**, 1009
setRolloverIcon, método de **AbstractButton**, 563
setSelected, método de **AbstractButton**, 631
setSelectionMode, método de **JList**, 572
setSize, método, 624
setStroke, método de **Graphics2D**, 535
setText
método, 611
método de **JLabel**, 554
método de **JTextArea**, 155
setTitle, método de **JFrame**, 618
setVerticalAlignment, método de **JLabel**, 553
setVerticalScrollBarPolicy, método de **JScrollPane**, 611
setVerticalTextPosition, método de **JLabel**, 554
setVisible, método de **Component**, 428, 592, 624
setVisibleRowCount, método de **JList**, 572
seudocódigo, 105
Shape, clase, 534
Short, clase, 433
ShortBuffer, clase, 810
show, método de **JPopupMenu**, 634
ShowDialog, método de **JColorChooser**, 518
showDocument, método de **AppletContext**, 827, 832
showInputDialog, 43
showMessageDialog, método de **JOptionPane**, 156
showMessageDialog, método de **JOptionPane**, 308
showMessageDialog, método de **JOptionPane**, 361
showMessageDialog, método de **JOptionPane**, 417
showMessageDialog, método de la clase **JOptionPane**, 38
showMessageDialog, método, 630
showOpenDialog, método, 776
showSaveDialog, método, 770
shutdown.sh, archivo, 1123
signo de moneda, $, 155
signo de porcentaje, 46
signo de suma (+), 490
signo mayor que, >, 83
signo menor que, <, 83
símbolo de MS-DOS, 32
símbolo de rombo pequeño, 178
Símbolo del sistema, 32
símbolos
de decisión, 108
de estado de acción, 106
de fusión, 113
de guardia, 108
especiales, 751
Simple, lenguaje, 984
SimpleGraph, programa de ejemplo del directorio **applets**, 74
simulación basada en software, 296
simulador de computadora, 296
sincronización de subprocesos, 708
SINGLE_INTERVAL_SELECTION, 572
SINGLE_SELECTION, 572

http://librosysolucionarios.net

Singleton, patrón de diseño de creación, 441
sintaxis de llamada al constructor de la superclase, 375
sistema
  binario (base 2), 477
  de administración de bases de datos (DBMS), 752, 1074
  de coordenadas, 510
  de menús, 608
  de reservaciones de una aerolínea, 285
  de ventanas, 550
  decimal (base 10), 477
  hexadecimal (base 16), 477
  octal (base 8), 477
  operativo UNIX, 7
sistemas de administración de bases de datos relacionados (RDBMS), 1074
sistemas numéricos, 477
sistemas operativos, 21
sitios Web, 11
size, método de BitSet, 1002
size, método de Map, 1066
skipBytes, método de DataInput, 754
sleep, método de Thread, 703
sobrecarga de métodos, 215
sobrescribir (redefinir) métodos, 81, 214, 358
socket, 827, 836
SocketChannel, clase, 897
SocketException, clase, 849
SocketImpl, clase, 909
SocketImplFactory, interfaz, 909
sockets
  de datagrama, 827
  de flujo, 827
  comunicaciones basadas en, 1112
software, 2
solicitante (método que hizo la llamada), 189
solicitarElevador
  mensaje del elevador, 278, 739
  operación del elevador, 228
sombreada (oculta), variable, 310
sombreado de un campo, 211
sonando, atributo del elevador, 134
sonarTimbre
  mensaje del elevador, 277
  operación del elevador, 227
sonido, imágenes, gráficos y video, 920
sort, método de Arrays, 1039
SortDemo, programa de ejemplo del directorio applets, 74
SortedMap, interfaz,1063
SOUTH
  región, 615
  constante de GridBagConstraints, 649
SOUTHEAST, constante de GridBagConstraints, 649
SOUTHWEST, constante de GridBagConstraints, 649
split, método de String, 491
spooler, 968
SpreadSheet, programa de ejemplo del directorio applets, 74
SQL, 1074
SQLException, clase, 1177
sqrt, método estático de Math, 190
Stack, clase, 998
StackTraceElement, clase, 692
start, método de JApplet, 81, 160, 214, 512, 702
startsWith, método de String, 459
startup.bat, shutdown.bat, startup.sh, archivos, 1123
Statement, interfaz, 1092
static
  campo, 330
  método, 39
  palabra reservada, 330, 332
stop
  método, 214
  método de Thread, 733
  método de Timer, 930
store, instrucción, 991
StreamTokenizer, clase, 987
String args[], 31
String getParameter(String nombre), método, 1117
String getServletInfo(), método, 1115
String
  clase, 435, 515, 755
  tipos, 1170
String[] getParameterValues (String nombre), método, 1117
StringIndexOutOfBoundsException, clase, 463
StringTokenizer, clase, 876, 990, 993
Stroke, interfaz, 535
subárbol
  derecho, 970
  izquierdo, 970
subclase, 79, 356
  de Estado, 445
subdirectorio webapps, 1124
sublista, 1048
subprocesamiento múltiple, 10, 22, 700
subproceso
  consumidor, 709
  de ejecución, 700
  despachador de eventos, 723
  main de ejecución, 705
  manejador de eventos, 723
  productor, 709
  recolector de basura, 701
SubprogramaBienvenido, 79
subprotocolo, 1092
subsistemas, 58, 909
substring, método de String, 463
Suma, 40
Sun One Studio 4 de Sun Microsystems, Inc, 1158
super, palabra reservada, 359, 375
superclase, 79, 314, 356
  directa, 356
  indirecta, 356
superclases abstractas, 397
suspend, resume, métodos desaprobados de Thread, 733
suspenderse, objeto Thread, 737
sustantivos, 94
SwingConstants, clase, 621
SwingConstants.BOTTOM, 554
SwingConstants.HORIZONTAL, 621
SwingConstants.LEFT, 553
SwingConstants.TOP, 554
SwingUtilities, clase, 638
switch, palabra reservada, 107
SymbolTest, programa de ejemplo del directorio applets, 74
synchronized, palabra clave, 709, 715
System (paquete java.lang), 333
System, clase, 225, 732, 753
System.err, 753, 754
System.exit(0), 428
System.gc, 385
System.in, System.out, 753
System.out, 32, 754
System.out.println, 32
SystemColor, clase, 534

T

Tab, 31
tabla de símbolos, 987
tablas, 1075
  de valores, arreglos de dos dimensiones, 270
TableModel, clase, 1094
taglib, directiva, 1171
tamaño de una variable, 45
tareas de preparación para la terminación, 329
TCP (Protocolo de control de transmisión), 827
tecla
  de acción, 584
  de dirección, Inicio, Fin, Re Pág, Av Pág, 584
  de función, Bloq Num, Impr Pant, Bloq Despl, Bloq Mayús y Pausa,), 584
  Fin, 621
teclas de método abreviado, 551
text/html, 1120
texto
  de plantilla fija, 1151
  seleccionado, 611
TexturePaint, clase, 510
theserverside.com, 1143
this
  palabra reservada, 209, 428
  referencia, 1154
Thread, clase, 702
Throwable, clase, 683
throws, cláusula, 680
TicTacToe, programa de ejemplo del directorio applets, 73, 74
tiempo compartido, 21
Tiempo1, clase, 304
Tiempo2, clase, 314
Tiempo3, clase, 319
tiempoRecorrido, atributo del elevador, 134
Timbre, clase del elevador, 95, 134, 277
Timer, clase, 926
TimesRoman, tipo de letra, 270
tipo
  de conjunto de resultados, 1097
  de una variable, 42, 45
  del atributo, 135
tipo-de-valor-de-retorno, 194
tipos
  de referencia, 91, 246
  integrados, 42
  no primitivos, 91
  predefinidos, 17
  primitivos, 42, 91, 246
tipos primitivos de Java
  boolean, 91
  byte, 91
  char, 91
  double, 91
  float, 91
  int, 91
  long, 91
  short, 91
tokens, 987
tolerante a fallas, 43
toLowerCase, método de String, 477
Tomcat, 1114
toString, método, 360
  de CharBuffer, 835
  de la clase Color, 523
  de Object, 393
  estático de Integer, 210
toUpperCase, método de Character, 464, 477
traca, petición de HTTP, 1117
transferencia de control, 105
transiciones entre estados, 177
transient, palabra reservada, 1001
translate, método de Graphics2D, 537
transmisión
  basada en flujos, orientada a la conexión, 849
  multiple (multicast), 870, 827
  sin conexión mediante datagramas, 849
transparencia, subclases de JComponent, 612
TreeMap, clase, 1063
TreeSet, clase, 1060
Tres en raya (Gato o Tic-Tac-Toe), 856
TresEnRaya, clase, 352
true, palabra reservada, 107, 611, 740
truncar
  cálculo, 124
  datos, 770
try
  bloque, 678, 831
  palabra reservada, 771
tryLock, método de FileChannel, 811
type, atributo, 1167
Types, clase, 1093
Types.INT, 1093

U

Ubicacion, clase del elevador, 435, 538, 740
ubicaciones en la memoria de la computadora, 45
UDP, el Protocolo de datagramas de usuario, 827
UEPS (último en entrar, primero en salir), 963
UIManager, clase, 638
UML
  Partners, 19
  nota, 656
Unicode, conjunto de caracteres, 751, 811
unidad
  de almacenamiento secundario, 21
  de entrada, 4, 21
  de memoria, 4, 21
  de salida, 4, 21
Unidad aritmética y lógica (ALU), 5, 21
Unidad central de procesamiento (CPU), 5, 21
uniones de línea, 535
UnknownHostException, clase, 837
UPDATE, instrucción, 1086
update, método de Component, 512, 612
update, método de Graphics, 215
updateComponentTreeUI, método estático de SwingUtilities, 638
URL, clase, 921
url-pattern, elemento de XML, 1125
URLs (Localizadores uniformes de recursos), 827, 1114
URLStreamHandler, clase, 910
uso de un búfer, 754

V

vaciado de la computadora, 297
valor
  centinela, 118
  clave, 263
  de prueba, 118
  de señal, 118
  de una variable, 45
  final, 146
valores
  enteros, 42
  predeterminados, 90
  RGB, 514
value, atributo, 1171
valueChanged, método, 831
valueOf, método de String, 466
variable
  de clase, 330
  de condición, 715
  de control, 114, 146
  de entorno CLASSPATH, 340
  de instancia, 90, 305
variables
  constantes, 207
  de instancia, 616
  de sólo lectura, 208, 252
  locales, 90, 115
Vector, clase, 828, 998
ventana, 624
  hija, 608
  padre, 608
ventana de comandos, 32
ventana de terminal o shell, 32
verbos, requerimientos de una nueva aplicación, 303
verdadera, condición, 49
verificar sus códigos de bytes, 13
vi, editor de texto, 12
viajar, operación del elevador, 228
vinculación
  dinámica, 407
  postergada, 407
vínculos Web, 11
vista, 656, 1042
  y el sentido, 550
VistaElevador, clase del elevador, 933
Visual Basic, 22
Visual Basic .NET, 10
Visual C++, 10, 22
visualización de la recursividad, 242
void, palabra reservada, 81, 194, 306
void addCookie(Cookie cookie), 1118
void destroy(), 1115
void init(ServletConfig config), 1115
void service(ServletRequest peticion, ServletResponse respuesta), 1115
void setContentType(String tipo), 1118
Volver a cargar, elemento del menú Subprograma de appletviewer, 75

W

wait, método de Thread, 703
.war, extensión de archivo, 1124
.wav (formato de archivo de onda de Windows), 930
Web
  servidor, 13
  sitios, 11
web.xml, archivo, 1124
web-app, elemento de XML, 1125
WEB-INF, directorio, 1124
WEB-INF/classes, directorio, 1124
WEB-INF/lib, directorio, 1124
weightx, campo de GrigBagConstraints, 648
weighty, campo de GridBagConstraints, 648
WEST, constante de GridBagConstraints, 649
WHERE, cláusula, 1081
while, palabra reservada, 148, 717, 843
widgets, 548
width, anchura de un subprograma, 83
windowActivated, método de WindowListener, 625
WindowAdapter, clase, 428
windowClosing
  método de WindowAdapter, 432
  método de WindowListener, 579, 625
WindowConstants, clase, 624
WindowConstants.DISPOSE_ON_CLOSE, 624
windowDeactivated, método de WindowListener, 625
windowDeiconified, método de WindowListener, 625
windowIconified, método de WindowListener, 625
WindowListener, interfaz, 579, 625
WindowOpened, método de WindowListener, 625
WireFrame, programa de ejemplo del directorio applets, 74
WORA (Write Once, Run Anywhere), 132
WritableByteChannel, interfaz, 811
write
  instrucción, 992
  método de FileChannel, 812
  método de OutputStream, 836
  método de WritableByteChannel, 811
writeBoolean, método de DataOutput, 754
writeByte, método de DataOutput, 754
writeBytes, método de DataOutput, 754
writeChar, método de DataOutput, 754
writeChars, método de DataOutput, 754, 784
writeDouble, método de DataOutput, 754, 784
writeFloat, método de DataOutput, 754
writeInt, método de DataOutput, 754
writeInt, método de DataOutput, 784
writeLong, método de DataOutput, 754
writeObject, método de ObjectOutput, 755, 771
Writer, clase, 753, 755
writeShort, método de DataOutput, 754
writeUTF, método de DataOutput, 754

Y

yield, método de Thread, 704

Z

.zip, extensión de archivo, 340

http://librosysolucionarios.net

http://libreria-universitaria.blogspot.com

Acuerdos de licencia para el usuario final

CONTRATO DE LICENCIA DEL CÓDIGO BINARIO DE SUN MICROSYSTEMS, INC.

LEA LAS CLÁUSULAS DE ESTE ACUERDO Y CUALQUIER CLÁUSULA SUPLEMENTARIA DE LA LICENCIA QUE SE PROPORCIONE (LO QUE COLECTIVAMENTE SE DENOMINARÁ COMO "ACUERDO") CUIDADOSAMENTE, ANTES DE ABRIR EL PAQUETE DE MEDIOS DE SOFTWARE. AL ABRIR ESTE PAQUETE DE MEDIOS DE SOFTWARE, USTED ACEPTA LAS CLÁUSULAS DE ESTE ACUERDO. SI ESTÁ ACCEDIENDO AL SOFTWARE EN FORMA ELECTRÓNICA, INDIQUE SU ACEPTACIÓN DE ESTOS TÉRMINOS SELECCIONANDO EL BOTÓN "ACEPTAR" AL FINAL DE ESTE ACUERDO. SI NO ESTÁ DE ACUERDO EN TODOS ESTOS TÉRMINOS, DEVUELVA INMEDIATAMENTE EL SOFTWARE SIN UTILIZAR AL LUGAR EN EL QUE LO COMPRÓ PARA QUE OBTENGA UN REEMBOLSO O, SI ESTÁ ACCEDIENDO AL SOFTWARE

EN FORMA ELECTRÓNICA, SELECCIONE EL BOTÓN "DECLINAR" AL FINAL DE ESTE ACUERDO.

1. LICENCIA DE USO. Sun le otorga una licencia no exclusiva e intransferible solamente para el uso interno del software y la documentación incluidos, y de cualquier corrección de error proporcionada por Sun (lo que colectivamente se denominará "Software"), según el número de usuarios y la clase de hardware de computadora para los que se haya pagado la cuota correspondiente.

2. RESTRICCIONES. El software es confidencial y está protegido por los derechos de autor. La titularidad del software y de todos los derechos de propiedad intelectual asociados es retenida por Sun y/o sus concesionarios. Con la excepción de lo que se autorice específicamente en cualquier Cláusula suplementaria de licencia, usted solamente podrá realizar una copia del Software con el fin de archivarlo. A menos que el cumplimiento de esta cláusula esté prohibido por las leyes aplicables, usted no podrá modificar, descompilar o aplicar ingeniería inversa al Software. Usted está de acuerdo en que el Software no está diseñado, ni se concede licencia o se pretende usar en el diseño, la construcción, operación y mantenimiento de ninguna instalación nuclear. Sun renuncia a cualquier garantía expresa o implícita de adecuación para tales usos. No se otorgan ningún derecho, título o interés en o sobre cualquier marca registrada, marca de servicio, logotipo o nombre comercial de Sun o sus concesionarios bajo este Contrato.

3. GARANTÍA LIMITADA. Sun le garantiza a usted por un periodo de noventa (90) días a partir de la fecha de compra, según lo demuestre una copia del recibo de compra, que los medios en los que se proporcione el Software (en caso de haberlos) estarán libres de defectos en materiales y mano de obra bajo un uso normal. Excepto lo antes mencionado, el Software se proporciona "COMO ESTÁ". Su único remedio y toda la responsabilidad de Sun bajo esta garantía limitada quedarán a elección de Sun, entre reemplazar los medios de Software o rembolsar la cuota pagada por el Software.

4. RENUNCIA DE GARANTÍA. A MENOS QUE SE ESPECIFIQUE DE OTRA FORMA EN ESTE ACUERDO, SE RENUNCIA A TODAS LAS CONDICIONES EXPRESAS O IMPLÍCITAS, REPRESENTACIONES Y GARANTÍAS, INCLUYENDO CUALQUIER GARANTÍA IMPLÍCITA DE FINES COMERCIALES, DE ADECUACIÓN PARA UN FIN ESPECÍFICO O INEXISTENCIA DE VIOLACIÓN, EXCEPTO EN LA EXTENSIÓN DE QUE ESTAS RENUNCIAS SE CONSIDEREN LEGALMENTE INVÁLIDAS.

5. LIMITACIÓN DE RESPONSABILIDAD. EN LA EXTENSIÓN NO PROHIBIDA POR LA LEY, EN NINGÚN CASO SUN Y SUS CONCESIONARIOS SERÁN RESPONSABLES POR LA PÉRDIDA DE INGRESOS, UTILIDADES O INFORMACIÓN, NI POR DAÑOS ESPECIALES, INDIRECTOS, CONSECUENTES, INCIDENTALES O REPRESIVOS, DE CUALQUIER TIPO QUE ESTOS SEAN OCASIONADOS, SIN IMPORTAR LA TEORÍA DE RESPONSABILIDAD QUE SUR-

http://librosysoluciones.net

JA DE, O QUE SE RELACIONE CON EL USO O LA INCAPACIDAD DE UTILIZAR EL SOFTWARE, INCLUSO SI SUN HA SIDO NOTIFICADO SOBRE LA POSIBILIDAD DE DICHOS DAÑOS. En ningún caso la responsabilidad de Sun para con usted, ya sea en contrato, agravio (incluyendo negligencia) o de cualquier otra forma, excederá la cantidad pagada por usted por el Software que está bajo este Contrato. Las limitaciones antes mencionadas se aplicarán incluso si la garantía antes mencionada no cumple su propósito esencial.

6. Terminación. Este Contrato permanecerá en vigor hasta la fecha en que sea terminado. Usted puede terminar este Contrato en cualquier momento, destruyendo todas las copias del Software. Este Contrato terminará inmediatamente sin notificación por parte de Sun si usted no cumple con alguna de las provisiones de este Contrato. Al momento de su terminación, usted deberá destruir todas las copias del Software.

7. Restricciones de exportación. Todo el Software e información técnica que se entreguen bajo este Acuerdo estarán sujetos a las leyes de control de exportaciones de los EE.UU., y puede estar sujeto a restricciones de exportación o importación en otros países. Usted está de acuerdo en cumplir estrictamente con dichas leyes y restricciones, y acepta que tiene la responsabilidad de obtener dichas licencias para exportar, reexportar o importar, según sea requerido, después de haber recibido el Software.

8. Derechos restringidos por el gobierno de los EE.UU. Si el Software está siendo adquirido por, o en representación de, el gobierno de los EE.UU. o por un contratista principal del gobierno de los EE.UU. (en cualquier nivel), entonces los derechos del gobierno sobre el Software y la documentación que lo acompaña serán según lo establecido en este Contrato; esto es en conformidad con la 48 CFR 227.7201 a 227.7202-4 (para las adquisiciones del Departamento de Defensa [DOD]) y con la 48 CFR 2.101 y 12.212 (para adquisiciones que no sean del DOD).

9. Ley gobernante. Cualquier acción relacionada con este Contrato estará gobernada por la ley de California y la ley federal estadounidense que la controla. No se aplicará ninguna elección de normas de derecho de ninguna jurisdicción.

10. Divisibilidad. Si alguna provisión de este Contrato fuese nula, este Contrato permanecerá en vigor sin tomar en cuenta esa provisión, a menos que esto frustre la intención de las partes, en cuyo caso este Contrato terminará inmediatamente.

11. Integración. Este Contrato es el acuerdo completo entre usted y Sun, en relación con su objeto. Sustituye todo tipo de comunicaciones, proposiciones y garantías orales o por escrito previas o contemporáneas, y prevalece sobre cualquier cláusula conflictiva o adicional de cualquier cita, orden, reconocimiento u otro tipo de comunicación entre las partes, en relación con su objeto durante el término de este Contrato. Ninguna modificación de este Contrato será obligatoria, a menos que sea por escrito y esté firmada por un representante autorizado de ambas partes.

CLÁUSULAS SUPLEMENTARIAS DE LICENCIA PARA EL KIT DE DESARROLLO DE SOFTWARE PARA JAVA™ 2, STANDARD EDITION

Estas cláusulas suplementarias de licencia ("Cláusulas suplementarias") complementan o modifican las cláusulas del Contrato de licencia del código binario (conocido colectivamente como el "Contrato"). Los términos en mayúsculas que no se definen en estas Cláusulas suplementarias tendrán el mismo significado que se les atribuye en el Contrato. Estas Cláusulas Suplementarias sustituirán a cualquier cláusula inconsistente o conflictiva en el Contrato, o en cualquier licencia contenida dentro del Software.

1. Concesión de licencia del software para desarrollo y uso interno. En sujeción a las cláusulas y condiciones de este Contrato, incluyendo pero sin limitarse a la Sección 4 (Restricciones sobre la tecnología Java) de estas Cláusulas Suplementarias, Sun le otorga a usted una licencia limitada no exclusiva e intransferible, sin cuotas, para reproducir y utilizar internamente el formato binario del Software completo y sin modificaciones, para el único fin de diseñar, desarrollar y probar sus applets y aplicaciones de Java que pretenda ejecutar en la plataforma Java ("Programas").

2. Licencia para distribuir software. En sujeción a las cláusulas y condiciones de este Contrato, incluyendo pero sin limitarse a, la Sección 4 (Restricciones sobre la tecnología Java) de estas Cláusulas Suplementarias, Sun le otorga a usted una licencia limitada no exclusiva e intransferible, sin cuotas, para reproducir y distribuir el Software, con la condición de que (i) usted distribuya el Software completo y sin modificaciones (a menos que se especifique de otra forma en el archivo README que se aplique) y que se incluya solamente como parte de, y para el único fin de ejecutar, sus Programas, (ii) los Programas agreguen una funcionalidad considerable y primordial al Software, (iii) usted no distribuya software adicional con la intención de reemplazar cualquiera de los componentes del Software (a menos que se especifique de otra forma en el archivo README que se aplique), (iv) usted no elimine ni altere ninguna leyenda o notificación de propiedad que estén contenidas en el Software, (v) usted distribuya solamente el Software sujeto a un contrato de licencia que proteja los intereses de Sun y que sea consistente con las cláusulas contenidas en este Contrato, y (vi) que usted esté de acuerdo en defender e indemnizar a Sun y sus concesionarios de y contra cualquier daño, costo, responsabilidad, monto de liquidación y/o gastos (incluyendo los honorarios de los abogados) incurridos en relación con cualquier reclamación, demanda o acción legal por parte de un tercero, que surja o resulte del uso o la distribución de cualquiera de los Programas y/o del Software.

(vi) incluya la siguiente declaración como parte de la documentación del producto (ya sea en copia impresa o en formato electrónico), como parte de una página de derechos de autor o página de notificación de derechos de propiedad, en un cuadro "Acerca de" o en cualquier otra forma razonablemente diseñada para hacer que la declaración sea visible para los usuarios del Software: "Este producto incluye código con licencia de RSA Security, Inc.", y (vii) que incluya la declaración "Ciertas porciones del Software con licencia de IBM están disponibles en http://oss.software.ibm.com/icu4j/".

3. Licencia para distribuir archivos redistribuibles. En sujeción a las cláusulas y condiciones de este Contrato, incluyendo pero sin limitarse a la Sección 4 (Restricciones sobre la tecnología Java) de estas Cláusulas Suplementarias, Sun le otorga a usted una licencia limitada no exclusiva e intransferible, sin cuotas, para reproducir y distribuir esos archivos específicamente identificados como redistribuibles en el archivo "README" del Software ("Redistribuibles") siempre y cuando: (i) usted distribuya los Redistribuibles completos y sin modificaciones (a menos que se especifique de otra forma en el archivo README que se aplique), y sólo los incluya como parte de los Programas, (ii) no distribuya software adicional con la intención de sustituir cualquier componente de los Redistribuibles (a menos que se especifique de otra forma en el archivo README que se aplique), (iii) no elimine ni altere ninguna leyenda o notificación de propiedad contenida en los Redistribuibles, (iv) sólo distribuya los Redistribuibles de conformidad con un contrato de licencia que proteja los intereses de SUN y que sea consistente con las cláusulas contenidas en el Contrato, (v) esté usted de acuerdo en defender e indemnizar a Sun y sus concesionarios de y contra cualquier daño, costo, responsabilidad, monto de liquidación y/o gastos (incluyendo los honorarios de los abogados) incurridos en relación con cualquier reclamación, demanda o acción legal por parte de un tercero, que surja o resulte del uso o la distribución de cualquiera de los Programas y/o del Software, (vi) incluya la siguiente declaración como parte de la documentación del producto (ya sea en copia impresa o en formato electrónico), como parte de una página de derechos de autor o página de notificación de derechos de propiedad, en un cuadro "Acerca de" o en cualquier otra forma razonablemente diseñada para hacer que la declaración sea visible para los usuarios del Software: "Este producto incluye código con licencia de RSA Security, Inc.", y (vii) que incluya la declaración "Ciertas porciones del Software con licencia de IBM están disponibles en http://oss.software.ibm.com/icu4j/".

4. Restricciones sobre la tecnología Java. Usted no puede modificar la Interfaz de la plataforma Java ("JPI", identificada como las clases contenidas dentro del paquete "java" o cualquier subpaquete del paquete "java"), creando clases adicionales dentro de la JPI o provocando de cualquier otra forma el complementar o modificar las clases en la JPI. En el caso en que usted cree una clase adicional y su(s) API(s) asociada(s), con lo cual (i) se extienda la funcionalidad de la plataforma Java, y (ii) se exponga a desarrolladores de software independientes para los fines de desarrollar software adicional que invoque a dicha(s) API(s) adicional(es), usted deberá publicar de inmediato una especificación precisa para dicha(s) API(s), de manera que todos los desarrolladores puedan utilizarla(s) libremente. Usted no puede crear, ni autorizar a sus concesionarios el crear, clases adicionales, interfaces o subpaquetes que se identifiquen de cualquier forma como "java", "javax", "sun" o cualquier convención similar, según lo especificado por Sun en cuanto a cualquier designación de convención de nomenclatura.

5. Notificación de actualizaciones automáticas de software de Sun. Usted acepta que el Software puede descargar, instalar y ejecutar automáticamente applets (subprogramas), aplicaciones, extensiones de software y versiones actualizadas del Software de Sun ("Actualizaciones de Software"), lo cual puede requerir que usted acepte cláusulas y condiciones actualizadas para la instalación. Si no se presentan cláusulas y condiciones adicionales durante la instalación, las Actualizaciones de Software se considerarán como parte del Software y estarán sujetas a las cláusulas y condiciones de este Contrato.

6. Notificación de descargas automáticas. Usted acepta que, al utilizar el Software y/o solicitar servicios que requieran el uso del mismo, el Software podrá descargar, instalar y ejecutar automáticamente aplicaciones de software de fuentes distintas de Sun ("Otro Software"). Sun no hace representaciones de una relación de cualquier tipo con los concesionarios de Otro Software. EN LA EXTENSIÓN NO PROHIBIDA POR LA LEY, EN NINGÚN CASO SUN Y SUS CONCESIONARIOS SERÁN RESPONSABLES POR LA PÉRDIDA DE INGRESOS, UTILIDADES O INFORMACIÓN, NI POR DAÑOS ESPECIALES, INDIRECTOS, CONSECUENTES, INCIDENTALES O REPRESIVOS, DE CUALQUIER FORMA QUE ÉSTOS SEAN OCASIONADOS, SIN IMPORTAR LA TEORÍA DE RESPONSABILIDAD QUE SURJA DE, O QUE SE RELACIONE CON EL USO O LA INCAPACIDAD DE UTILIZAR OTRO SOFTWARE, INCLUSO SI SUN HA SIDO NOTIFICADO SOBRE LA POSIBILIDAD DE DICHOS DAÑOS.

7. Distribución por editoriales. Esta sección se aplica a su distribución del Software con su libro o revista impresos (según se utilicen esos términos comúnmente en la industria) en relación con la tecnología de Java ("Publicación"). En sujeción a, y con la condición de que usted cumpla con las restricciones y obligaciones contenidas en el Contrato, además de la licencia otorgada en el Párrafo 1 anterior, Sun por este medio le otorga un derecho limitado no exclusivo e intransferible para reproducir copias completas y sin modificaciones del Software en medios electrónicos (los "Medios") para el único fin de incluirlos y distribuirlos con su(s) Publicación(es), de acuerdo con las siguientes cláusulas: (i) Usted no podrá distri-

http://librosysolucionarios.net

buir el Software en forma independiente; éste deberá distribuirse con su(s) Publicación(es); (ii) Usted es responsable de descargar el Software del sitio Web de Sun aplicable; (iii) Usted debe referirse al Software como Kit de desarrollo de software para JavaTM 2, Standard Edition, versión 1.4.0; (iv) El Software deberá reproducirse en su totalidad y sin ninguna modificación (incluyendo, sin limitación, la Licencia de código binario y las Cláusulas suplementarias de licencia que acompañan al Software, junto con las notificaciones de derechos de propiedad contenidas en el Software); (v) La etiqueta de los Medios deberá incluir la siguiente información: Copyright 2002, Sun Microsystems, Inc. Todos los derechos reservados, Su uso está sujeto a las cláusulas de la licencia. Sun, Sun Microsystems, el logtipo de Sun, Solaris, Java, el logotipo Java Coffee Cup, J2SE y todas las marcas comerciales y logotipos basados en Java son marcas comerciales o marcas registradas de Sun Microsystems, Inc. en los EE.UU. y en otros países. Esta información deberá colocarse en la etiqueta de los Medios, de tal manera que se aplique solamente al Software de Sun; (vi) Usted deberá identificar claramente el Software como producto de Sun en la caja o la etiqueta de los Medios, y usted no podrá declarar ni suponer que Sun sea responsable de cualquier software de terceros que esté contenido en los Medios; (vii) Usted no podrá incluir software de terceros en los Medios con la intención de reemplazar o sustituir al Software; (viii) Usted deberá indemnizar a Sun por todos los daños que surjan si usted no cumple con los requerimientos de este Contrato. Además, usted deberá defender, a costa suya, todas y cualquier reclamación contra Sun por terceras partes, y deberá pagar todos los daños concedidos por una corte de jurisdicción competente, o dicho monto de liquidación negociado por usted, producido por, o en conexión con, el uso, reproducción o distribución que usted haga del Software y/o la Publicación. Su obligación de proporcionar indemnización bajo esta sección deberá producirse siempre y cuando que Sun: (i) le proporcione a usted una pronta notificación de la reclamación; (ii) le otorgue a usted el control completo de la defensa y solución de la reclamación; (iii) le proporcione a usted, a costa suya, toda la información, asistencia y autoridad disponibles para defenderse; y (iv) no se haya comprometido o arreglado dicha reclamación sin que usted lo haya autorizado previamente por escrito; y (ix) Usted deberá proporcionar a Sun una notificación escrita para cada Publicación; dicha notificación deberá incluir la siguiente información: (1) título de la Publicación, (2) autor(es), (3) fecha de la Publicación y (4) números ISBN o ISSN. Dicha notificación deberá enviarse a Sun Microsystems, Inc., 4150 Network Circle, M/S USCA12-110, Santa Clara, California 95054, U.S.A , Attention: Contracts Administration.

8. Marcas registradas y logotipos, Usted reconoce y está de acuerdo que Sun es propietario de las marcas registradas SUN, SOLARIS, JAVA, JINI, FORTE e iPLANET, y de todas las marcas registradas, marcas de servicio, logotipos y demás designaciones de marcas relacionadas con SUN, SOLARIS, JAVA, JINI, FORTE e iPLANET ("Marcas de Sun"), y está de acuerdo en cumplir con los Requerimientos de uso de la marca registrada y el logotipo de Sun, que se encuentran actualmente en http://www.sun.com/policies/trademarks. Cualquier uso que usted haga de las Marcas de Sun se aplicará a beneficio de Sun.

9. Código fuente. El Software puede contener código fuente que se proporciona únicamente para fines de referencia, de conformidad con los términos de este Contrato. El código fuente no podrá redistribuirse, a menos que esto se indique expresamente en este Contrato.

10. Terminación por infracción: Cualquiera de las partes podrá terminar este Contrato inmediatamente en caso de que cualquier parte del Software se convierta, o que en opinión de cualquiera de las partes sea probable que se convierta, en el sujeto de una reclamación por infracción de un derecho de propiedad intelectual.

Para cualquier consulta, póngase en contacto con: Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A (LFI#111374/Form ID#011801).

SUN™ ONE STUDIO 4 UPDATE 1, COMMUNITY EDITION CLÁUSULAS SUPLEMENTARIAS DE LA LICENCIA

Estas cláusulas suplementarias de licencia ("Cláusulas suplementarias") complementan o modifican las cláusulas del Contrato de licencia del código binario (conocido colectivamente como el "Contrato"). Los términos en mayúsculas que no se definen en estas Cláusulas suplementarias tendrán el mismo significado que se les atribuye en el Contrato. Estas Cláusulas Suplementarias sustituirán a cualquier cláusula inconsistente o conflictiva en el Contrato, o en cualquier licencia contenida dentro del Software.

1. Concesión de licencia del software para desarrollo y uso interno. En sujeción a las cláusulas y condiciones de este Contrato, incluyendo pero sin limitarse a, la Sección 4 (Restricciones sobre la tecnología Java) de estas Cláusulas Suplementarias, Sun le otorga a usted una licencia limitada no exclusiva e intransferible, para reproducir y utilizar internamente el formato binario del Software completo y sin modificaciones, para el único fin de diseñar, desarrollar y probar sus applets y aplicaciones ("Programas"). En caso de que usted esté diseñando, desarrollando y probando applets y aplicaciones de Java para una versión específica de la plataforma Java, cualquier salida ejecutable generada por un compilador que esté contenido en el Software deberá (a) compilarse solamente a partir de código fuente que se conforme a la versión correspondiente de la Especificación de lenguaje Java para OEM (OEM Java Language Specification); (b) estar en el formato de archivo de clase definido por la versión correspondiente de la Especificación de máquina virtual de Java para OEM (OEM Java Virtual Machine Specification); y (c) ejecutarse apropiadamente en un entorno de tiempo de ejecución de referencia, según lo especificado con Sun, asociado con dicha versión de la plataforma Java.

2. Licencia para distribuir software. En sujeción a las cláusulas y condiciones de este Contrato, incluyendo pero sin limitarse a, la Sección 4 (Restricciones sobre la tecnología Java) de estas Cláusulas Suplementarias, Sun le otorga a usted una licencia limitada no exclusiva e intransferible, para reproducir y distribuir el Software solamente en formato de código binario, con la condición de que (i) usted distribuya el Software completo y sin modificaciones, (ii) usted no distribuya software adicional con la intención de reemplazar cualquiera de los componentes del Software, (iii) si va a distribuir applets y aplicaciones de Java para una versión específica de la plataforma Java, cualquier salida ejecutable generada por un compilador que esté contenido en el Software deberá (a) compilarse solamente a partir de código fuente que se conforme a la versión correspondiente de la Especificación de lenguaje Java para OEM (OEM Java Language Specification); (b) estar en el formato de archivo de clase definido por la versión correspondiente de la Especificación de máquina virtual de Java para OEM (OEM Java Virtual Machine Specification); y (c) ejecutarse apropiadamente en un entorno de tiempo de ejecución de referencia, según lo especificado con Sun, asociado con dicha versión de la plataforma Java, (iv) usted no elimine ni altere ninguna leyenda o notificación de propiedad que estén contenidas en el Software, (v) usted distribuya solamente el Software sujeto a un contrato de licencia que proteja los intereses de Sun y que sea consistente con las cláusulas contenidas en este Contrato, y (vi) que usted esté de acuerdo en defender e indemnizar a Sun y sus concesionarios de y contra cualquier daño, costo, responsabilidad, monto de liquidación y/o gastos (incluyendo los honorarios de los abogados) incurridos en relación con cualquier reclamación, demanda o acción legal por parte de un tercero, que surja o resulte del uso o la distribución de cualquiera de los Programas y/o del Software.

3. Licencia para distribuir archivos redistribuibles. En sujeción a las cláusulas y condiciones de este Contrato, incluyendo pero sin limitarse a, la Sección 4 (Restricciones sobre la tecnología Java) de estas Cláusulas Suplementarias, Sun le otorga a usted una licencia limitada no exclusiva e intransferible, para reproducir y distribuir el formato binario de esos archivos específicamente identificados como redistribuibles en el archivo "RELEASE NOTES" del Software ("Redistribuibles") siempre y cuando: (i) usted distribuya los Redistribuibles completos y sin modificaciones (a menos que se especifique de otra forma en el archivo RELEASE NOTES que se aplique), y sólo los incluya como parte de los Programas, (ii) no distribuya software adicional con la intención de sustituir cualquier componente de los Redistribuibles, (iii) no elimine ni altere ninguna leyenda o notificación de propiedad contenida en los Redistribuibles, (iv) si va a distribuir applets y aplicaciones de Java para una versión específica de la plataforma Java, cualquier salida ejecutable generada por un compilador que esté contenido en el Software deberá (a) compilarse solamente a partir de código fuente que se conforme a la versión correspondiente de la Especificación de lenguaje Java para OEM (OEM Java Language Specification); (b) estar en el formato de archivo de clase definido por la versión correspondiente de la Especificación de máquina virtual de Java para OEM (OEM Java Virtual Machine Specification); y (c) ejecutarse apropiadamente en un entorno de tiempo de ejecución de referencia, según lo especificado con Sun, asociado con dicha versión de la plataforma Java, (v) usted sólo distribuya los Redistribuibles de conformidad con un contrato de licencia que proteja los intereses de SUN y que sea consistente con las cláusulas contenidas en el Contrato, y (vi) esté usted de acuerdo en defender e indemnizar a Sun y sus concesionarios de y contra cualquier daño, costo, responsabilidad, monto de liquidación y/o gastos (incluyendo los honorarios de los abogados) incurridos en relación con cualquier reclamación, demanda o acción legal por parte de un tercero, que surja o resulte del uso o la distribución de cualquiera de los Programas y/o del Software.

4. Restricciones sobre la tecnología Java. Usted no puede modificar la Interfaz de la plataforma Java ("JPI", identificada como las clases contenidas dentro del paquete "java" o cualquier subpaquete del paquete "java"), creando clases adicionales dentro de la JPI o provocando de cualquier otra forma el complementar o modificar las clases en la JPI. En el caso en que usted cree una clase adicional y su(s) API(s) asociada(s), con lo cual (i) se extienda la funcionalidad de la plataforma Java, y (ii) se exponga a desarrolladores de software independientes para los fines de desarrollar software adicional que invoque a dicha(s) API(s) adicional(es), usted deberá publicar inmediatamente una especificación precisa para dicha(s) API(s), de manera que todos los desarrolladores puedan utilizarla(s) libremente. Usted no puede crear, ni autorizar a sus concesionarios el crear, clases adicionales, interfaces o subpaquetes que se identifiquen de cualquier forma como "java", "javax", "sun" o cualquier convención similar, según lo especificado por Sun en cuanto a cualquier designación de convención de nomenclatura.

http://librosysolucionarios.net

5. Disponibilidad del entorno en tiempo de ejecución de Java. Consulte la versión apropiada de la licencia del código binario del Entorno en tiempo de ejecución de Java (actualmente ubicada en http://www.java.sun.com/jdk/index.html) la disponibilidad del código del entorno en tiempo de ejecución, el cual podrá distribuirse con applets y aplicaciones de Java.

6. Distribución por editoriales. Esta sección se aplica a su distribución del Software con su libro o revista impresos (según se utilicen esos términos comúnmente en la industria) en relación con la tecnología de Java ("Publicación"). En sujeción a, y con la condición de que usted cumpla con las restricciones y obligaciones contenidas en el Contrato, además de la licencia otorgada en el Párrafo 1 anterior, Sun por este medio le otorga un derecho limitado no exclusivo e intransferible para reproducir copias completas y sin modificaciones del Software en medios electrónicos (los "Medios") para el único fin de incluirlos y distribuirlos con su(s) Publicación(es), de acuerdo con las siguientes cláusulas: (i) Usted no podrá distribuir el Software en forma independiente; éste deberá distribuirse con su(s) Publicación(es); (ii) Usted es responsable de descargar el Software del sitio Web de Sun aplicable; (iii) Usted debe referirse al Software como Sun ONE Studio 4, Community Edition; (iv) El Software deberá reproducirse en su totalidad y sin ninguna modificación (incluyendo, sin limitación, la Licencia de código binario y las Cláusulas suplementarias de licencia que acompañan al Software, junto con las notificaciones de derechos de propiedad contenidas en el Software); (v) La etiqueta de los Medios deberá incluir la siguiente información: Copyright 2002, Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054. Java, SUN One y todas las marcas registradas y logotipos basados en Java y Sun One son marcas registradas de Sun Microsystems, Inc. en los EE.UU. y otros países. Esta información deberá colocarse en la etiqueta de los Medios, de tal manera que se aplique solamente al Software de Sun; (vi) Usted deberá identificar claramente el Software como producto de Sun en la caja o la etiqueta de los Medios, y usted no podrá declarar ni suponer que Sun sea responsable de cualquier software de terceros que esté contenido en los Medios; (vii) Usted no podrá incluir software de terceros en los Medios con la intención de reemplazar o sustituir al Software o que represente una competencia directa para el Software; (viii) usted deberá indemnizar a Sun por todos los daños que surjan si usted no cumple con los requerimientos de este Contrato. Además, usted deberá defender, a costa suya, todas y cualquier reclamación contra Sun por terceras partes, y deberá pagar todos los daños concedidos por una corte de jurisdicción competente, o dicho monto de liquidación negociado por usted, producido por, o en conexión con, el uso, reproducción o distribución que usted haga del Software y/o la Publicación. Su obligación de proporcionar indemnización bajo esta sección deberá producirse siempre y cuando que Sun: (i) le proporcione a usted una pronta notificación de la reclamación; (ii) le otorgue a usted el control completo de la defensa y solución de la reclamación; (iii) le proporcione a usted, a costa suya, toda la información, asistencia y autoridad disponibles para defenderse; y (iv) no se haya comprometido o arreglado dicha reclamación sin que usted lo haya autorizado previamente por escrito; (ix) usted deberá proporcionar a Sun una notificación escrita para cada Publicación; dicha notificación deberá incluir la siguiente información: (1) título de la Publicación, (2) autor(es), (3) fecha de la Publicación y (4) números ISBN o ISSN. Dicha notificación deberá enviarse a Sun Microsystems, Inc., 4150 Network Circle, M/S USCA12-110, Palo Alto, CA 94303-4900, Attention: Contracts Administration; y (x) usted deberá proporciona a Sun informes trimestrales por escrito en relación con el número de copias del Software distribuido durante el trimestre anterior; dichos informes deberán enviarse a Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA, 95054, Attn.: Sun ONE Studio Product Management Group, M/S UOAK01.

7. Marcas registradas y logotipos. Usted reconoce y está de acuerdo que Sun es propietario de las marcas registradas SUN, SOLARIS, JAVA, JINI, FORTE e iPLANET, y de todas las marcas registradas, marcas de servicio, logotipos y demás designaciones de marcas relacionadas con SUN, SOLARIS, JAVA, JINI, FORTE e iPLANET ("Marcas de Sun"), y está de acuerdo en cumplir con los Requerimientos de uso de la marca registrada y el logotipo de Sun, que se encuentran actualmente en http://www.sun.com/policies/trademarks. Cualquier uso que usted haga de las Marcas de Sun se aplicará a beneficio de Sun.

8. Código fuente. El Software puede contener código fuente que se proporciona únicamente para fines de referencia, de conformidad con los términos de este Contrato. El código fuente no podrá redistribuirse, a menos que esto se indique expresamente en este Contrato.

9. Terminación por infracción: Cualquiera de las partes podrá terminar este Contrato inmediatamente en caso de que cualquier parte del Software se convierta, o que en opinión de cualquiera de las partes sea probable que se convierta, en el sujeto de una reclamación por infracción de un derecho de propiedad intelectual.

Para cualquier consulta, póngase en contacto con: Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054.

(LFI#117241/Form ID#011801)

IBM® CLOUDSCAPE™ SERVER EDITION, VERSIÓN 5.0.4. CONTRATO DE LICENCIA INTERNACIONAL PARA LA EVALUACIÓN DE LOS PROGRAMAS

Parte 1: Condiciones generales

POR FAVOR LEA ESTE CONTRATO CUIDADOSAMENTE ANTES DE UTILIZAR EL PROGRAMA. IBM SÓLO LE CONCEDERÁ LA LICENCIA DE USO DEL PROGRAMA SI ANTES ACEPTA LAS CONDICIONES DE ESTE ACUERDO. CON EL USO DEL PROGRAMA EL CLIENTE MANIFIESTA SU ACUERDO CON ESTAS CONDICIONES. SI NO ACEPTA LAS CONDICIONES DE ESTE ACUERDO, DEBERÁ DEVOLVER INMEDIATAMENTE A IBM EL PROGRAMA SIN QUE ÉSTE HAYA SIDO USADO.

El Programa es propiedad de International Business Machines Corporation, de alguna de sus subsidiarias (IBM) o de un proveedor de IBM, está sujeto a derechos de autor y de licencia de uso y por tanto no se transmite su propiedad.

El término Programa hace referencia al programa original y a sus copias totales o parciales. Un Programa consta de instrucciones legibles por máquina, sus componentes, datos, contenido audiovisual (tales como imágenes, texto, grabaciones o dibujos), así como los materiales bajo licencia asociados.

Este Acuerdo incluye la Parte 1: Condiciones Generales y la Parte 2: Condiciones exclusivas de cada país, y constituye el acuerdo completo en lo que al uso del Programa se refiere, y sustituye cualquier comunicación verbal o escrita previa entre el Cliente e IBM. Los términos de la Parte 2 pueden sustituir o modificar a las de la Parte 1.

1. Licencia

Uso del Programa.

IBM le concede una licencia de uso del Programa no exclusiva e intransferible.

Usted puede 1) utilizar el Programa sólo con fines de evaluación interna, prueba o para propósitos de demostración, ya sea como prueba o como prueba con opción a compra y 2) hacer e instalar un número razonable de copias del Programa como soporte del tal uso, a menos que IBM especifique un número determinado de copias en la documentación del Programa. Los términos de la licencia son de aplicación a cada copia que usted haga. Deberá reproducir el aviso de derechos de autor y cualquier otro aviso relativo a la titularidad en cada copia, o copia parcial, del Programa.

EL PROGRAMA PUEDE CONTENER UN DISPOSITIVO DE DESACTIVACIÓN QUE IMPEDIRÁ SU USO UNA VEZ FINALIZADA LA LICENCIA. USTED NO PODRÁ MANIPULAR ESTE DISPOSITIVO DE DESACTIVACIÓN NI EL PROGRAMA. ADEMÁS DEBERÁ TOMAR PRECAUCIONES PARA EVITAR PÉRDIDA DE DATOS QUE PUEDA RESULTAR CUANDO EL PROGRAMA NO PUEDA SER YA UTILIZADO.

Usted deberá 1) mantener un registro de todas las copias del Programa y 2) asegurarse de que cualquier persona que usa el Programa lo haga sólo para su uso autorizado y de acuerdo con los términos del presente Acuerdo. Usted no puede: 1) utilizar, copiar, modificar o distribuir el Programa de un modo distinto al especificado en este Acuerdo; 2) desensamblar, descompilar o traducir el Programa de un modo distinto al permitido específicamente por la legislación del país en el que se está utilizando el programa, y que no tenga la posibilidad de renuncia contractual; 3) sublicenciar, alquilar o arrendar el Programa.

La licencia será efectiva cuando el Cliente haga el primer uso del Programa y finalizará 1) en la fecha de terminación que se especifique en la documentación que acompaña al Programa, o 2) cuando el Programa automáticamente se desactive. Salvo que IBM le autorice, en la documentación del Programa, a retener el Programa (en cuyo caso serán de aplicación los cargos que correspondan), usted deberá destruir el Programa y todas las copias hechas del mismo dentro de los 10 días siguientes a la fecha de finalización de la licencia.

2. Exclusión de Garantía

EXCEPTO LAS QUE SEAN OBLIGATORIAS POR LEY, IBM NO DARÁ NINGUNA GARANTÍA, EXPRESA O IMPLÍCITA, INCLUYENDO PERO SIN LIMITARSE A, GARANTÍAS DE NO CUMPLIMIENTO Y GARANTÍAS IMPLÍCITAS DE COMERCIALIZACIÓN O IDONEIDAD PARA UNA FINALIDAD DETERMINADA, EN LO REFERENTE AL PROGRAMA O AL SOPORTE TÉCNICO SI LO HUBIERA. IBM NO DARÁ GARANTÍA ALGUNA RESPECTO A LA CAPACIDAD DEL PROGRAMA PARA QUE PROCESE CORRECTAMENTE, SUMINISTRE Y/O RECIBA DATOS DE FECHAS DENTRO Y ENTRE LOS SIGLOS XX Y XXI.

Esta exclusión de garantía es también de aplicación a los subcontratistas de IBM, proveedores de IBM o desarrolladores de Programas (colectivamente llamados "Proveedores").

Los fabricantes, distribuidores o editores de Programas No IBM pueden proporcionar sus propias garantías.

3. Limitación de Responsabilidad

NI IBM NI LOS DISTRIBUIDORES DE IBM SERÁN RESPONSABLES DE LOS DAÑOS DIRECTOS O INDIRECTOS, INCLUYENDO PERO SIN LIMITARSE A, PÉRDIDA DE BENEFICIOS, PÉRDIDA DE ECONOMÍAS PREVISTAS, O CUALQUIER DAÑO INCIDENTAL, ESPECIAL O DAÑO ECONÓMICO DERIVADO, INCLUSO SI IBM HUBIERA SIDO ADVERTIDA DE LA POSIBILIDAD DE QUE OCURRIESEN. ALGUNAS JURISDICCIONES NO PERMITEN LA EXCLUSIÓN O LIMITACIÓN DE DAÑOS INCIDENTALES O DERIVADOS, EN CUYO CASO PODRÍA SER QUE LA LIMITACIÓN O EXCLUSIÓN ANTERIOR NO LE SEA APLICABLE.

http://librosysoluciones.net

4. General

Ningún derecho estatutario de los consumidores podrá ser cancelado o limitado por este Acuerdo.

IBM podrá dar por finalizada esta licencia si usted incumple las condiciones de este Acuerdo. Si IBM finaliza su licencia, usted deberá destruir inmediatamente el Programa y todas las copias hechas del mismo.

Usted no podrá exportar el Programa.

Ni usted ni IBM podrán interponer acción legal alguna a que pudiera dar lugar este Acuerdo transcurridos más de dos años desde que se produjo su causa, a no ser que se establezca de otra manera por la legislación del país sin la posibilidad de renuncia contractual o limitación. Ni usted ni IBM serán responsables del incumplimiento de sus obligaciones cuando sea debido a causas de fuerza mayor. No habrá ningún cargo adicional por el uso del Programa por la duración de esta licencia. IBM no proporciona servicios de programa ni soporte técnico, a menos que IBM lo especifique de otra manera. Este Acuerdo se rige por la legislación del país en el que haya adquirido el Programa, excepto 1) en Australia, las leyes del Estado o Territorio en el que se realizó la transacción rigen este Acuerdo; 2) en Albania, Armenia, Bielorrusia, Bosnia/Herzegovina, Bulgaria, Croacia, República Checa, Georgia, Hungría, Kazakhstan, Kirghizia, Antigua República Yugoslava de Macedonia, Moldavia, Polonia, Rumania, Rusia, República Federal de Yugoslavia, República de Eslovaquia, Eslovenia y Ucrania, las leyes de Austria rigen este Acuerdo; 3) en el Reino Unido, todas las controversias relacionadas con este Acuerdo se regirán por la Ley Inglesa y serán sometidas a la jurisdicción exclusiva de los tribunales Ingleses; 4) en Canadá, las leyes de la Provincia de Ontario rigen este Acuerdo, Y 5) en los Estados Unidos y en Puerto Rico, y en la República Popular de China, las leyes del Estado de Nueva York rigen este Acuerdo.

Parte 2: Condiciones exclusivas de cada país

AUSTRALIA:

Exclusión de garantía (Sección 2):

Se añade el párrafo siguiente a esta Sección:

Aunque IBM especifica que no existe garantía alguna, usted podría tener determinados derechos en virtud de la Ley de 1974 de Prácticas comerciales (Trade Practices Act 1974) u otra legislación y están sólo limitados hasta lo permitido por la legislación aplicable.

Limitación de responsabilidad (Sección 3):

Se añade el párrafo siguiente a esta sección:

Cuando IBM incumpla una condición o garantía que se derive de la Ley de 1974 de Prácticas comerciales (Trade Practices Act 1974), la responsabilidad de IBM está limitada a la reparación o sustitución de las mercancías o al suministro de mercancías equivalentes. Cuando dicha condición o garantía esté relacionada con el derecho de venta, la posesión pacífica o el derecho libre, o las mercancías sean de una clase que generalmente se adquiere para uso o consumo familiar, doméstico o personal, no se aplicará ninguna de las limitaciones de este párrafo.

ALEMANIA:

Exclusión de garantía (Sección 2):

Se añaden los párrafos siguientes a esta sección:

El periodo de garantía mínimo para los Programas es de seis meses.

En el caso de que un Programa se suministre sin Especificaciones, IBM únicamente garantiza que la información del Programa describe el Programa correctamente y que éste puede utilizarse de acuerdo con lo que se especifica en la información del Programa. Usted es el responsable de que el Programa se utilice según la información del Programa dentro del periodo de la garantía que permite la devolución del dinero.

Limitación de responsabilidad (Sección 3):

Se añade el párrafo siguiente a esta sección:

Las limitaciones y exclusiones especificadas en el Acuerdo no se aplicarán a los daños causados por IBM por dolo o negligencia grave y por garantía expresa.

INDIA:

General (Sección 4):

El párrafo siguiente sustituye al cuarto párrafo de esta Sección:

Si no se entabla ningún pleito o acción legal, durante los dos años posteriores a la aparición de la causa de la acción, respecto a cualquier demanda de una de las partes contra la otra, los derechos de la parte interesada en relación con tal demanda se perderán quedando la otra parte libre de sus obligaciones respecto a dicha demanda.

IRLANDA:

Exclusión de garantía (Sección 2):

Se añade el párrafo siguiente a esta sección:

Salvo lo expresamente indicado en estos términos y condiciones, todas las condiciones estatutarias, incluyendo todas las garantías implícitas, pero sin prejuicio de la generalidad de todas las garantías precedentes que se deriven de la Ley de 1893 sobre Venta de Mercancías (Sale of Goods Act 1893) o de la Ley de 1980 sobre Venta de Mercancías y Suministro de Servicios (Sale of Goods and Supply of Services Act 1980), quedan excluidas por el presente documento.

ITALIA:

Limitación de responsabilidad (Sección 3):

Esta sección es reemplazada por la siguiente:

Salvo expresamente previsto por ley, IBM no se hace responsable por daño alguno que pudiera producirse.

NUEVA ZELANDA:

Exclusión de garantía (Sección 2):

Se añade el párrafo siguiente a esta sección:

Aunque IBM especifica que no existe garantía alguna, el Cliente podría tener determinados derechos en virtud de la Ley de 1993 de Prácticas Comerciales (Consumer Guarantees Act 1993) u otra legislación que no puede excluirse ni limitarse. La Ley de 1993 sobre Garantías del Consumidor (Consumer Guarantees Act 1993) no se aplicará a ningún bien o servicio que proporcione IBM, si el Cliente precisa dichos bienes y servicios para realizar un negocio, como se define en dicha Ley.

Limitación de responsabilidad (Sección 3):

Se añade el párrafo siguiente a esta sección:

Cuando los Programas no se adquieran para las finalidades de una empresa, tal como se define en la Ley de 1993 sobre Garantías del Consumidor (Consumer Guarantees Act 1993), las limitaciones de esta Sección están sujetas a las limitaciones de dicha Ley.

REINO UNIDO:

Limitación de responsabilidad (Sección 3):

Se añade el párrafo siguiente a esta sección al final del primer párrafo:

La limitación de responsabilidad no se aplicará en ningún caso a un incumplimiento de las obligaciones de IBM que se deriven del Artículo 12 de la Ley de Ventas de Mercancías de 1979 (Sale of Goods Act 1979) o del Artículo 2 de la Ley de Suministro de Mercancías y Servicios de 1982 (Supply of Goods and Services Act 1982).

Z125-5543-01 (10/97)

INFORMACIÓN SOBRE LA LICENCIA

Los Programas que se enumeran a continuación son programas bajo licencia sujetos a los siguientes términos y condiciones además de los del Acuerdo Internacional de Licencia para la Evaluación de Programas.

Nombre de programa:

IBM Cloudscape Personal Edition Version 5.0.4

Entorno Operativo Especificado

La información sobre las especificaciones del Programa y sobre el entorno operativo especificado se pueden encontrar en la documentación que acompaña al Programa, tal como la Guía de instalación / del usuario.

Periodo de evaluación

La licencia comienza en la fecha en que se utiliza por primera vez el Programa y finaliza después de 60 días.

Condiciones únicas del Programa

Derechos restringidos para usuarios del gobierno de los EE.UU.

Derechos restringidos para usuarios del gobierno de los EE.UU: el uso, la duplicación o divulgación están restringidos por el Contrato de itinerario GSA ADP con IBM Corporation.

IBM, DB2 y Cloudscape son marcas registradas de IBM Corporation en los Estados Unidos, en otros países o en ambos.

Java y todas las marcas registradas y logotipos de Java, además de Solaris, son marcas registradas de Sun Microsystems, Inc. en los Estados Unidos, en otros países o en ambos.

Microsoft, Windows y Windows NT son marcas registradas de Microsoft Corporation en los EE.UU., en otros países o en ambos.

UNIX es una marca registrada en los Estados Unidos, en otros países o en ambos, y su licencia se otorga exclusivamente a través de X/Open Company Limited.

LICENCIA DEL SOFTWARE APACHE, VERSIÓN 1.1

Copyright (c) 2000 The Apache Software Foundation. Todos los derechos reservados.

La redistribución y el uso del software Apache en los formatos fuente y binario, con o sin modificación, están permitidos siempre y cuando se cumplan las condiciones siguientes:

1. Las redistribuciones del código fuente deben contener el aviso de copyright anterior, esta lista de condiciones y la renuncia siguiente.

2. Las redistribuciones en formato binario deben reproducir el aviso de copyright anterior, esta lista de condiciones y la renuncia siguiente en la documentación y/o en otros materiales proporcionados con la distribución.

3. La documentación para el usuario final incluida con la redistribución, si la hay, debe incluir la siguiente frase: "Este producto incluye software desarrollado por The Apache Software Foundation (www.apache.org)." Alternativamente, esa información puede aparecer en el software, si dicha información de terceros aparece normalmente y, en tal caso, en el sitio donde aparece.

http://librosysolucionarios.net

4. Los nombres "Apache" y "Apache Software Foundation" no pueden utilizarse para respaldar ni promocionar productos derivados de este software sin permiso previo por escrito. Para obtener permiso por escrito, póngase en contacto con apache@apache.org.

5. Los productos derivados de este software no pueden denominarse "Apache", ni puede aparecer la palabra "Apache" en su nombre sin el permiso previo por escrito de The Apache Software Foundation.

ESTE SOFTWARE APACHE SE PROPORCIONA "TAL CUAL" Y SE RENUNCIA A TODA GARANTÍA EXPLÍCITA O IMPLÍCITA, INCLUIDAS, ENTRE OTRAS, LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD E IDONEIDAD PARA UN PROPÓSITO DETERMINADO. EN NINGÚN CASO THE APACHE SOFTWARE FOUNDATION NI SUS COLABORADORES SERÁN RESPONSABLES DE NINGUNA RECLAMACIÓN DIRECTA, INDIRECTA, FORTUITA, ESPECIAL, EXONERANTE O DAÑOS DERIVADOS (INCLUIDOS, PERO SIN LIMITARSE A, LA OBTENCIÓN DE BIENES O SERVICIOS SUSTITUTOS, LA PÉRDIDA DE UTILIZACIÓN, DATOS O BENEFICIOS O LA INTERRUPCIÓN DEL FUNCIONAMIENTO DEL NEGOCIO) INDEPENDIENTEMENTE DE CÓMO SE HAYAN PRODUCIDO Y DE CUALQUIER RESPONSABILIDAD, YA SEA POR CONTRATO, RESPONSABILIDAD ESTRICTA O AGRAVIO (INCLUIDAS LA NEGLIGENCIA U OTROS) POR DAÑOS OCASIONADOS POR EL USO DE ESTE SOFTWARE, INCLUSO SI SE HUBIERA AVISADO DE LA POSIBILIDAD DE QUE SE PRODUJERA DICHO DAÑO.

Este software consta de contribuciones voluntarias efectuadas por diversos particulares en nombre de The Apache Software Foundation. Para obtener más información sobre The Apache Software Foundation, consulte www.apache.org.

Partes de este software están basadas en software de dominio público, originalmente escrito en el Centro nacional para aplicaciones de súper cómputo de la Universidad de Illinois, Urbana-Champaign.

CONTRATO DE LICENCIA Y GARANTÍA LIMITADA DE PRENTICE HALL

LEA CUIDADOSAMENTE LAS SIGUIENTES CLÁUSULAS Y CONDICIONES ANTES DE ABRIR ESTE PAQUETE DE SOFTWARE. ESTE DOCUMENTO LEGAL ES UN ACUERDO ENTRE USTED Y PRENTICE HALL, INC. (LA "COMPAÑÍA".) AL ABRIR ESTE PAQUETE DE SOFTWARE SELLADO, USTED MANIFIESTA ESTAR DE ACUERDO EN QUEDAR OBLIGADO POR ESTAS CLÁUSULAS Y CONDICIONES. SI NO ESTÁ DE ACUERDO CON ESTAS CLÁUSULAS Y CONDICIONES, NO ABRA EL PAQUETE DE SOFTWARE. DEVUELVA INMEDIATAMENTE EL PAQUETE DE SOFTWARE SIN ABRIR, JUNTO CON TODOS LOS ARTÍCULOS QUE LO ACOMPAÑAN, AL LUGAR EN EL QUE LOS OBTUVO PARA QUE RECIBA UN REEMBOLSO COMPLETO DE CUALQUIER SUMA QUE HAYA PAGADO.

1. OTORGAMIENTO DE LICENCIA. En consideración por haber comprado usted el libro, y teniendo en cuenta que está usted de acuerdo en cumplir con las cláusulas y condiciones de este Contrato, la Compañía le otorga a usted el derecho no exclusivo de utilizar y mostrar la copia del programa de software incluido (que en lo sucesivo se le denominará el "SOFTWARE") en una sola computadora (es decir, con un solo CPU) y en una sola ubicación, siempre y cuando usted cumpla con las cláusulas de este Contrato. La Compañía se reserva todos los derechos que no le sean otorgados a usted expresamente en este Contrato.

2. PROPIEDAD DEL SOFTWARE: Usted posee solamente los medios magnéticos o físicos (los medios incluidos) en los que el SOFTWARE está grabado o fijado, pero la Compañía y los desarrolladores de software retienen todos los derechos, título y propiedad del SOFTWARE grabado en la(s) copia(s) original(es) de los medios y en todas las copias subsecuentes del SOFTWARE, sin importar el formato o los medios en los que puedan existir el original u otras copias. Esta licencia no es una venta del SOFTWARE original, o copia del mismo, a usted.

3. RESTRICCIONES DE COPIA. Este SOFTWARE y los materiales impresos que lo acompañan, junto con el manual de usuario (la "Documentación") están sujetos al derecho de autor. Los programas individuales en los medios están protegidos por las leyes de derecho de autor que corresponden a los autores de cada programa. Algunos de los programas en los medios incluyen contratos de licencia por separado. Si usted tiene la intención de utilizar uno de estos programas, debe leer y seguir su contrato de licencia que lo acompañe. Usted no puede copiar la Documentación o el SOFTWARE, excepto cuando haga una sola copia del SOFTWARE para fines de respaldo o archivo solamente. Usted será legalmente responsable por cualquier infracción de copia o derechos de autor que sea provocada o se de cómo consecuencia al hecho de que usted no cumpla con las cláusulas de esta restricción.

4. RESTRICCIONES DE USO: Usted no podrá instalar en red el SOFTWARE ni utilizarlo de cualquier otra forma en más de una computadora o terminal de computadora al mismo tiempo. Usted puede transferir físicamente el SOFTWARE de un equipo a otro, siempre y cuando el SOFTWARE se utilice solamente en una computadora a la vez. Usted no puede distribuir copias del SOFTWARE o la Documentación a otros. Usted no puede aplicar ingeniería inversa, desensamblar, descompilar, modificar, adaptar, traducir o crear trabajos derivativos con base en el SOFTWARE o la Documentación sin el previo consentimiento por escrito de la Compañía.

5. RESTRICCIONES DE TRANSFERENCIA: La licencia del SOFTWARE incluido se otorga solamente a usted, y no puede transferirse a nadie más sin el previo consentimiento por escrito de la Compañía. Cualquier transferencia no autorizada del SOFTWARE provocará la terminación inmediata de este Contrato.

6. TERMINACIÓN: Esta licencia permanecerá en vigor hasta la fecha en que sea terminada. Esta licencia terminará automáticamente sin notificación por parte de la Compañía, siendo nula e inválida si usted no cumple con las provisiones o limitaciones de esta licencia. Al momento de su terminación, usted deberá destruir la Documentación y todas las copias del SOFTWARE. Todas las provisiones de este Contrato en relación con las garantías, limitación de responsabilidad, remedios o daños y nuestros derechos de propiedad permanecerán aún después de la terminación.

7. MISCELÁNEOS. Este Contrato deberá interpretarse de conformidad con las leyes de los Estados Unidos de América y el Estado de Nueva York, y deberá beneficiar a la Compañía, sus afiliados y cesionarios.

8. GARANTÍA LIMITADA Y RENUNCIA DE GARANTÍA: La Compañía garantiza que el SOFTWARE, cuando sea utilizado apropiadamente y de conformidad con la Documentación, operará en conformidad substancial con la descripción del SOFTWARE establecida en la Documentación. La Compañía no garantiza que el SOFTWARE cumplirá con los requerimientos de usted, o que la operación del SOFTWARE será ininterrumpida o libre de errores. La Compañía garantiza que los medios en los que se entrega el SOFTWARE estarán libres de defectos en materiales y mano de obra bajo el uso normal, durante un periodo de treinta (30) días a partir de la fecha en que usted lo compró. Su único remedio y la única obligación de la Compañía bajo estas garantías limitadas es, a elección de la Compañía, la devolución del artículo garantizado por un reembolso de cualquier cantidad que usted haya pagado, o el reemplazo del artículo. Cualquier reemplazo de SOFTWARE o medios bajo las garantías no deberá extenderse más allá del periodo de garantía original. La garantía limitada establecida anteriormente no se aplicará al SOFTWARE que la Compañía determine, en buena fe, que haya sido objeto de mal uso, negligencia, instalación inapropiada, reparación, alteración, o daños ocasionados por usted. CON LA EXCEPCIÓN DE LAS GARANTÍAS EXPRESAS ESTABLECIDAS ANTERIORMENTE, LA COMPAÑÍA RENUNCIA A TODAS LAS GARANTÍAS, EXPRESAS O IMPLÍCITAS, INCLUYENDO SIN LIMITACIÓN LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD Y ADECUACIÓN PARA UN PROPÓSITO ESPECÍFICO. CON LA EXCEPCIÓN DE LA GARANTÍA EXPRESA ESTABLECIDA ANTERIORMENTE, LA COMPAÑÍA NO GARANTIZA NI HACE REPRESENTACIÓN ALGUNA EN RELACIÓN CON EL USO O LOS RESULTADOS DEL USO DEL SOFTWARE EN TÉRMINOS DE SU EXACTITUD, PRECISIÓN, CONFIABILIDAD, ACTUALIDAD O DE CUALQUIER OTRA FORMA.

EN NINGÚN CASO LA COMPAÑÍA O SUS EMPLEADOS, AGENTES, PROVEEDORES O CONTRATISTAS SERÁN RESPONSABLES POR CUALQUIER DAÑO INCIDENTAL, INDIRECTO, ESPECIAL O CONSECUENTE QUE SURJA DE, O EN CONEXIÓN CON LA LICENCIA OTORGADA BAJO ESTE CONTRATO, O POR LA PÉRDIDA DE USO, DE INFORMACIÓN, DE INGRESOS O UTILIDADES, O DE CUALQUIER OTRO TIPO DE PÉRDIDAS, DERIVADAS COMO RESULTADO DE LESIONES A CUALQUIER PERSONA, O LA PÉRDIDA O DAÑO A LOS BIENES, O RECLAMACIONES DE TERCERAS PERSONAS, INCLUSO SI LA COMPAÑÍA O UN REPRESENTANTE AUTORIZADO DE LA COMPAÑÍA HA SIDO NOTIFICADO DE LA POSIBILIDAD DE DICHOS DAÑOS. EN NINGÚN CASO LA RESPONSABILIDAD DE LA COMPAÑÍA POR LOS DAÑOS CON RESPECTO AL SOFTWARE SE EXCEDERÁ DE LOS MONTOS QUE USTED HAYA PAGADO, EN CASO DE HABERLOS, POR EL SOFTWARE.

ALGUNAS JURISDICCIONES NO PERMITEN LA LIMITACIÓN DE GARANTÍAS IMPLÍCITAS O RESPONSABILIDAD POR DAÑOS INCIDENTALES, INDIRECTOS, ESPECIALES O CONSECUENTES, POR LO QUE TAL VEZ LAS ANTERIORES LIMITACIONES NO SIEMPRE SE APLIQUEN. LAS GARANTÍAS EN ESTE CONTRATO LE OTORGAN A USTED CIERTOS DERECHOS LEGALES ESPECÍFICOS, Y TAMBIÉN PUEDE TENER OTROS DERECHOS QUE PUEDEN VARIAR DE ACUERDO CON LAS LEYES LOCALES.

RECONOCIMIENTO

USTED RECONOCE QUE HA LEÍDO ESTE CONTRATO, QUE LO HA COMPRENDIDO Y ESTÁ DE ACUERDO EN REGIRSE POR SUS CLÁUSULAS Y CONDICIONES. USTED TAMBIÉN ESTÁ DE ACUERDO EN QUE ESTE CONTRATO ES LA DECLARACIÓN COMPLETA Y EXCLUSIVA DEL ACUERDO ENTRE USTED Y LA COMPAÑÍA, Y QUE SUSTITUYE A TODAS LAS PROPOSICIONES O CONTRATOS ANTERIORES, YA SEA EN FORMA ORAL O ESCRITA, Y A CUALQUIER OTRO TIPO DE COMUNICACIONES ENTRE USTED Y LA COMPAÑÍA O CUALQUIER REPRESENTANTE DE LA COMPAÑÍA, EN RELACIÓN CON EL OBJETO DE ESTE CONTRATO.

En caso de que tenga dudas o preguntas en relación con este Contrato, o si desea ponerse en contacto con la Compañía por cualquier motivo, hágalo por escrito a la siguiente dirección:

Robin Short
Prentice Hall PTR
One Lake Street
Upper Saddle River, New Jersey 07458

http://librosysolucionarios.net

Acuerdo de licencia y garantía limitada

El software se proporciona "COMO ESTÁ", sin ninguna garantía de ninguna clase. Ni los autores, los desarrolladores de software, ni Pearson Educación hacen ninguna representación o garantía, ya sea expresa o implícita, con respecto a los programas de software, su calidad, precisión o adecuación para un propósito específico. Por lo tanto, ni los autores, desarrolladores de software ni Pearson Educación tendrán responsabilidad alguna para con usted o cualquier otra persona o entidad con respecto a cualquier pérdida o daños, supuestos o reales, ocasionados directa o indirectamente por los programas contenidos en el CD. Esto incluye, pero no se limita a, la interrupción del servicio, pérdida de datos, pérdida de tiempo en el salón de clases, pérdida de utilidades por consultoría o de otro tipo, o daños consecuentes derivados del uso de estos programas. Si el CD en sí está defectuoso, usted podrá devolverlo para obtener un reemplazo. El uso de este software está sujeto a las cláusulas y condiciones de la Licencia del Código Binario que se incluye en este libro. Lea las licencias cuidadosamente. Al abrir este paquete, usted está de acuerdo en regirse por las cláusulas y condiciones de estas licencias. Si no está de acuerdo, no abra el paquete.

Consulte el acuerdo de licencia para el usuario final que viene en el CD-ROM para obtener más detalles.

Uso del CD-ROM

La interfaz con el contenido de este CD-ROM está diseñada para iniciar automáticamente. Si no aparece una pantalla de inicio automáticamente cuando usted inserte el CD en su computadora, explore el disco y haga doble clic en el archivo `AUTORUN.EXE`.

Contenido del CD-ROM

- Kit de desarrollo para la plataforma Java™ 2, Standard Edition, versión 1.4.1 para Windows y Linux (32 bits).
- Sun™ ONE Studio 4, Community Edition para plataformas Windows y Linux.
- IBM® Cloudscape™ Server Edition, versión 5.0.4 (prueba de evaluación de 60 días).
- Apache Tomcat versión 4.1.12 de Apache Software Foundation.
- Ejemplos de código activo (Live Code™) de este libro.
- Recursos Web: vínculos hacia sitios de Internet mencionados en el libro.
- Recursos adicionales en formato PDF (de Adobe® Acrobat®) no incluidos en el libro.
- Todo el código fuente utilizado en los ejercicios del libro.
- Archivos de texto *en español*, en formato PDF, con material no incluido en el libro.

Requerimientos de software y hardware del sistema

- Procesador Pentium III de 500 MHz (mínimo) o superior.
- Microsoft Windows® NT (con Service Pack 6a), Windows 2000 Professional (con Service Pack 3 o superior), Windows XP, o
- Red Hat Linux 7.2.
- 256 MB de RAM (mínimo), 512 MB de RAM (recomendado).
- Unidad de CD-ROM.
- Conexión a Internet y explorador Web.

PROGRAMAS EDUCATIVOS, S.A. DE C.V.
CALZ. CHABACANO NO. 65,
COL. ASTURIAS, DELG. CUAUHTEMOC,
C.P. 06850, MEXICO, D.F.

EMPRESA CERTIFICADA POR EL
INSTITUTO MEXICANO DE NORMALIZACION
Y CERTIFICACION A.C. BAJO LAS NORMAS
ISO-9002: 1994/NMX-CC-004/1995
CON EL NO. DE REGISTRO RSC-044
Y ISO-14001:1996/NMX-SAA-001/1998 IMNC
CON EL NO. DE REGISTRO RSAA-052

http://librosysolucionarios.net