Presentación

Android es un OS (*Operating System* o Sistema Operativo en castellano) Open Source pensado para teléfonos móviles y desarrollado por la **Open Handset Alliance** (OHA) bajo autorización de Google. La OHA se compone de alrededor de 80 empresas, tales como Samsung, HTC, SFR, Orange, Asus, Qualcomm...

Android se basa en un kernel Linux y se distribuye bajo una licencia **Apache License 2.0**. Separa la capa hardware de la capa lógica, es decir, cualquier teléfono Android puede ejecutar la misma aplicación y, de este modo, se puede crear un amplio abanico de posibilidades para los fabricantes, los usuarios y los desarrolladores.

Android es también un framework y, como desarrollador, tendrá acceso al SDK (*Software Development Kit* - Kit de desarrollo) y a todo el código fuente de la plataforma. Esto le permitirá comprender su funcionamiento, crear versiones personalizadas del OS y, por supuesto, desarrollar sus propias aplicaciones.

Por consiguiente, varios dispositivos exóticos han visto la luz con Android: radios de coche, relojes, cascos para esquiar, etc.

Origen

La historia de Android empieza en octubre de 2003, cuando Andy Rubin, Rich Miner, Nick Sears y Chris White crearon la empresa Android en Palo Alto (California).

Google compró la empresa en agosto de 2005. Dos años más tarde, se anunció la Open Handset Alliance y Android pasa a ser oficialmente Open Source.

La primera versión del SDK Android 1.0 aparece en 2008 con el primer teléfono con Android (HTC Dream).

En abril de 2009, la versión 1.5 (API 3) de Android vio la luz. Esta versión, bautizada como **Cupcake**(Pastelito), inaugura los nuevos nombres de versión de Android.

Las siguientes versiones siguieron este patrón:

- **Donut** (Donut) 1.6 (API 4): septiembre de 2009
- **Eclair** (Rayo) 2.0/2.1 (API 7): octubre de 2009
- **Froyo** (Yogur congelado) 2.2.x (API 8): mayo de 2010
- **Gingerbread** (Pan de gengibre) 2.3.x (API 10): diciembre de 2010
- **Honeycomb** (Panal de abejas) 3.x (API 11 - 12 - 13): febrero de 2011
- **Ice Cream Sandwich** (Sandwich de helado) 4.0.x (API 14 - 15): octubre de 2011. Esta versión, utilizada en este libro, pasa a ser la rama principal de Android. Unificó la versión para tablet (Honeycomb) y la versión para smartphone (Gingerbread).
- **Jelly Bean** (Judía de gominola) 4.1.x (API 16 y +): junio de 2012. Las novedades de esta versión se presentarán a lo largo de este libro.

El icono de Android se llama **Bugdroid**. Este pequeño robot verde tiene como origen, según rumores, el personaje de un videojuegos de los años 1990 para Atari: **Gauntlet: The Third Encounter**.

Android es, también:

- 900 000 activaciones por día.
- 400 000 millones de productos Android activados en el mundo.
- 600 000 aplicaciones disponibles en Google Play.
- 1 500 millones de descargas de aplicaciones al mes.

Google Play

Para permitir a los usuarios acceder a un gran surtido de aplicaciones, Android utiliza un market llamado **Google Play** (anteriormente llamado Android Market).

Permite a los usuarios:

- Buscar y descargar aplicaciones, libros, canciones y películas.
- Puntuar, comentar, desinstalar y actualizar las aplicaciones que ya se han instalado en el dispositivo.

Como desarrollador, publicar una aplicación en Google Play es muy sencillo y se realiza en pocos pasos (véase el capítulo Principios de programación - Exportación y certificación de una aplicación):

- Exportación de la aplicación.
- Certificación de la aplicación.
- Envío al market.

1. Creación de una cuenta de desarrollador

Para publicar una aplicación, necesita una cuenta de desarrollador de Android.

La creación de una cuenta de desarrollador requiere:

- Una dirección de Gmail válida.
- Un número de teléfono.
- 25 $ (alrededor de 20 €).

Conéctese a https://play.google.com/apps/publish/ para crear su propia cuenta.

Una vez haya creado la cuenta, podrá:

- Publicar una nueva aplicación.
- Seguir el estado de sus aplicaciones ya publicadas (comentarios, puntuaciones, estadísticas, errores, etc.).
- Eliminar aplicaciones ya publicadas.
- Actualizar sus aplicaciones (versión, descripción, etc.).

El identificador de una aplicación Android en el market no es su nombre sino el identificador de su package. Varias aplicaciones pueden tener un nombre similar, pero sólo el identificador del package debe ser único (ver capítulo Mi primera aplicación: HelloAndroid).

2. Publicación de una aplicación

La publicación de una aplicación se realiza en la dirección siguiente:https://market.android.com/publish/Home

Para iniciar el envío de una aplicación, haga clic en el botón **Subir aplicación**. Un pop-up permite seleccionar el archivo correspondiente a la aplicación que se desea publicar.

La nueva versión del market permite añadir archivos adicionales a su aplicación si ésta no sobrepasa los 50 MB. Esta opción es muy útil para los videojuegos, por ejemplo.

Cuando finaliza el envío, un cuadro de diálogo le permite introducir algunos datos sobre la aplicación.

Entonces podrá:

- Guardar los datos y pasar a la siguiente etapa.
- Reemplazar la aplicación seleccionada.
- Eliminar la aplicación seleccionada.

En la siguiente etapa hay que añadir algunos datos sobre su aplicación:

- Un mínimo de dos capturas de pantalla.
- Un icono de la aplicación en alta definición (512 x 512).
- Los idiomas soportados por la aplicación.
- Un título y una descripción para la aplicación en cada idioma soportado.
- El tipo de aplicación (juego o aplicación).
- La categoría de la aplicación (libro, financiera, multimedia, social, etc.).
- El tipo de público al que va dirigida (todo el mundo, público concreto, etc.).
- Los países en los que la aplicación estará disponible.
- Aceptar los términos y condiciones de Google Play.

Una vez se ha introducido toda esta información, podrá:

- Guardar los cambios sin publicar la aplicación.
- Guardar los cambios y publicar la aplicación.

La duración de la publicación de una aplicación es de algunos minutos. La comprobación se realiza a posteriori. Si los usuarios indican que una aplicación es maliciosa, se retira del market y de todos los dispositivos en los que se encuentre.

3. Seguimiento y actualización de una aplicación

- Para realizar el seguimiento de sus aplicaciones, conéctese a:https://play.google.com/apps/publish/

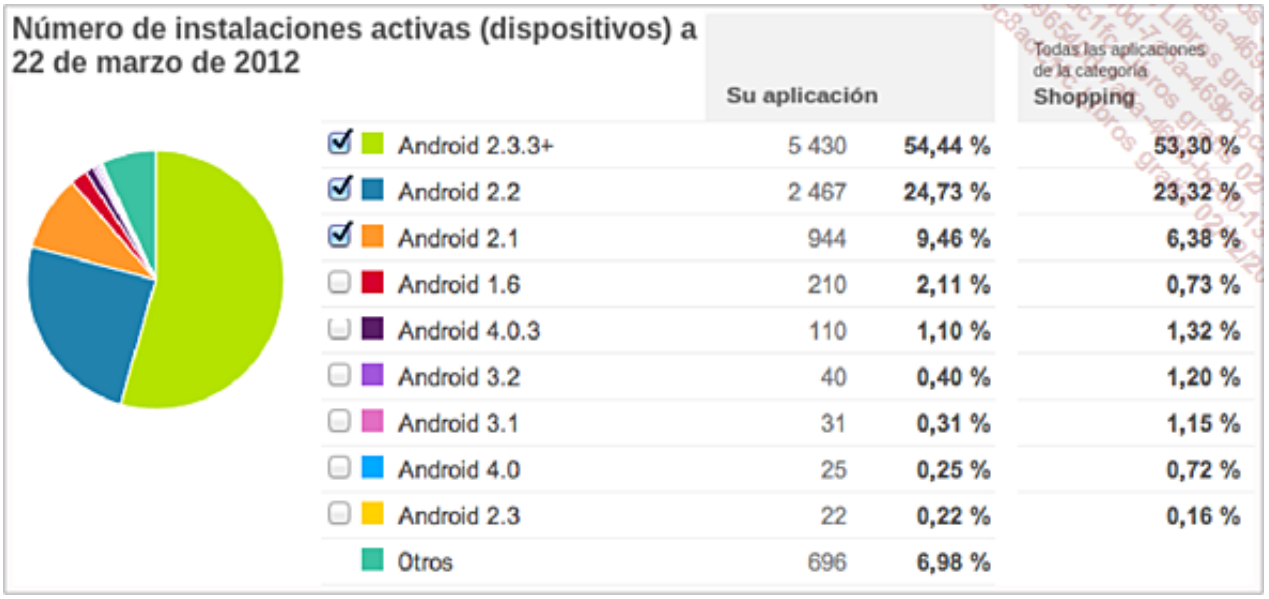
Un panel muestra una lista de todas sus aplicaciones, con estadísticas para cada una de ellas.

Se puede acceder a la siguiente información:

- Nombre de la aplicación.
- Su versión.
- Su tipo y su categoría.
- Su puntuación media.
- Número de instalaciones totales.
- Número de instalaciones activas (significa que la aplicación está todavía presente en los dispositivos).
- Su precio.
- Errores reportados por los usuarios.
- El estado de la aplicación.

El botón **Comments** (comentarios) permite mostrar los detalles de las puntuaciones otorgadas a la aplicación así como los comentarios asociados.

El botón **Statistics** (estadísticas) permite visualizar gráficos de estadísticas sobre su aplicación (gráfico de instalación, instalación por versión de Android, instalación por tipo de dispositivo, instalación por país e instalación por idioma).



- Si hace clic en el nombre de una aplicación, accederá a una página que le permite publicar una actualización.

Esta página permite visualizar toda la información ya introducida. Es entonces cuando se puede:

- Modificar estos datos.

- Eliminar la aplicación.
- Guardar los cambios.
- Publicar los cambios.

Puede enviar una nueva versión de su aplicación en cualquier momento. Una vez enviada la nueva versión, deberá completar el campo **Recent Changes** (Últimos cambios) para cada idioma soportado para informar al usuario sobre las novedades aportadas por la nueva versión de la aplicación.

Instalación del entorno Java

El primer requisito para desarrollar en Android es la instalación de un entorno Java y, concretamente, del **JDK** (*Java Development Kit* - Kit de desarrollo Java).

- Para instalar (si todavía no estuviera instalado) un entorno Java, vaya a la siguiente página del sitio web de Oracle: http://www.oracle.com/technetwork/java/javase/downloads/index.html

  Descargue e instale la versión del JDK (en este caso la versión 6.38) que se corresponda con su OS.

SDK Android

1. Presentación

La siguiente etapa permite instalar y configurar el SDK Android. Este SDK contiene todas las herramientas necesarias para crear una aplicación Android. Está disponible para Windows, Mac OS y Linux.

Cada versión del SDK contiene:

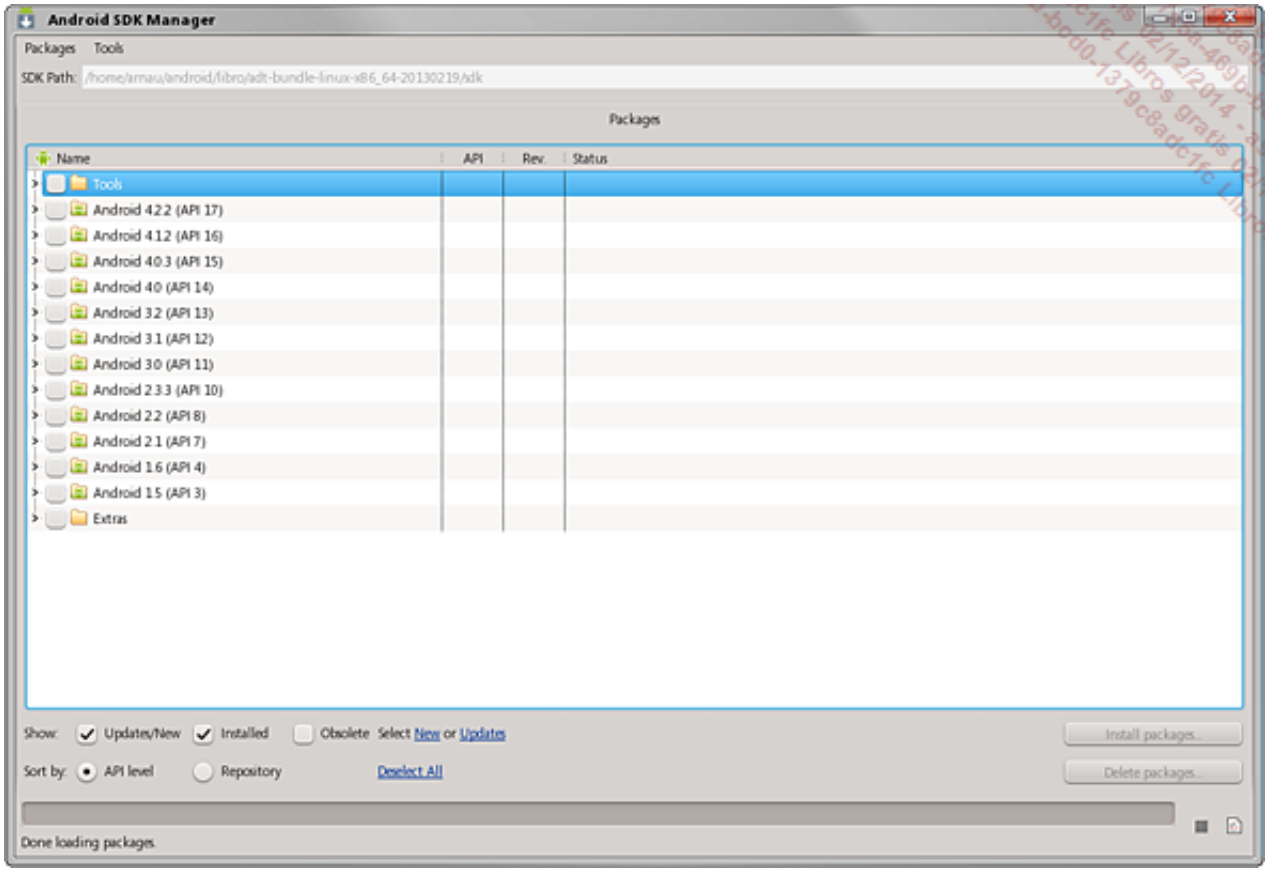
- **aapt - Android Asset Packaging Tool**: esta herramienta sirve para crear y analizar archivos *.apk (Android Package) (véase el capítulo Principios de programación - Principios generales). Estos archivos contienen el programa de su aplicación.
- **adb - Android Debug Bridge**: el objetivo de esta herramienta es establecer conexiones con un teléfono Android o un emulador facilitando el acceso a su terminal. Todo ello para listar su contenido o ejecutar comandos. Esta herramienta también sirve para transferir una aplicación o archivos a un teléfono Android o a un emulador.
- **dx - Dalvik Cross-Assembler**: esta herramienta sirve para fusionar y convertir archivos de clases estándar Java (*.class) en archivos binarios (*.dex). Estos archivos se pueden ejecutar en la VM Dalvik (véase el capítulo Principios de programación - Principios generales).
- **ddms - Dalvik Debug Monitor Service** (véase el capítulo Depuración y gestión de errores - Principios): esta herramienta se utiliza en la depuración de aplicaciones y permite:
  - Hacer captura de pantallas.
  - Ver los threads en ejecución.
  - Ver el Logcat.
  - Averiguar la lista de procesos en ejecución en el dispositivo.
  - Simular el envío de mensajes y llamadas.
  - Simular una localización, etc.
- El SDK también contiene un sistema que permite crear y gestionar emuladores así como la documentación para cada versión de Android y ejemplos para cada una de las API.

2. Instalación

Para instalar el SDK Android, diríjase a la siguiente dirección:http://developer.android.com/sdk/index.html, y descargue e instale la versión del SDK correspondiente a su OS.

En Windows: evite tener espacios en las rutas de su SDK.

Una vez ha terminada la instalación, ejecute el SDK Android (ejecutable **SDK Manager** en Windows y binario **android** presente en la carpeta **Tools** para Mac y Linux). Debería abrirse la siguiente ventana:



Esta herramienta, **Android SDK Manager**, le permitirá actualizar su SDK, instalar nuevas versiones de Android o actualizar las versiones ya instaladas.

Se procederá a instalar, como mínimo, los siguientes elementos:

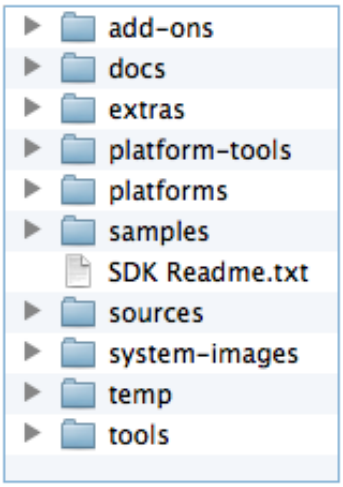
- Todo el contenido de la sección **Tools**:
  - **SDK Tools** (instalado por defecto): contiene los elementos necesarios para desarrollar, probar y depurar su aplicación. Se instala por defecto con el SDK Android y se actualizará cuando aparezcan nuevas versiones.
  - **SDK Plateform-tools**: contiene herramientas de desarrollo, de pruebas y de depuración para su aplicación. Estas herramientas se adaptan a la versión de Android utilizada por su aplicación.
- En la sección **Android 4.0.3** y **4.1**:
  - **Documentation Android**: la documentación de la versión de Android seleccionada.
  - **SDK Plateform**: la versión del SDK.
  - **Sample**: ejemplos de las API.
  - **Google APIs**: la versión del SDK Android incluyendo las APIs de Google (Google Maps, por ejemplo).
  - **Source**: los fuentes de esta versión de Android.
- En la sección **Extras**, únicamente:
  - **Android Support Package**: anteriormente llamado "compatibility package" (paquete de compatibilidad). Representa una librería que se incluirá en el proyecto Android para utilizar funcionalidades no incluidas en el framework Android (tales como el **ViewPager**) o las funcionalidades no disponibles en todas las versiones de Android (tales como los**Fragments**).
  - **Google USB Driver Package**: necesario si se dispone a desarrollar en Windows con un

teléfono Google (Nexus one, Nexus S o Galaxy Nexus). Contiene los drivers para desarrollar, probar y depurar en estos teléfonos.

A continuación, haga clic en el botón **Install Packages** (Instalar paquetes) para finalizar la instalación.

3. Utilización

Una vez se ha instalado el SDK, podrá explorar su estructura de directorios:



- La carpeta **add-ons** contiene las distintas extensiones o SDK que ha descargado fuera del SDK principal (SDK tablet Sony y Google APIs, por ejemplo).
- La carpeta **docs** contiene la documentación de Android.
- La carpeta **extras** contiene las herramientas que ha descargado (sección Extras en el SDK Manager).
- La carpeta **platforms** contiene las distintas versiones del SDK que se ha descargado, organizadas por número de versión.
- La carpeta **platform-tools** contiene herramientas que le permiten desarrollar y depurar sus aplicaciones.
- La carpeta **samples** contiene un surtido de ejemplos de las APIs de Android.
- La carpeta **sources** contiene los fuentes de Android en función de las versiones del SDK.
- La carpeta **tools** contiene herramientas que facilitan el uso del SDK Android.
- **AVDManager** es el gestor de emuladores (únicamente presente en Windows).
- **SDKManager** es el gestor del SDK (únicamente presente en Windows).

Eclipse

1. Presentación

El segundo elemento necesario para desarrollar en Android es Eclipse. Este programa es un IDE (*Integrated Development Environment* - Entorno de desarrollo integrado) que facilita el desarrollo y la creación de aplicaciones.

Eclipse es el IDE utilizado en este libro, porque se integra más fácilmente con el SDK Android. También puede utilizar Netbeans o cualquier otro IDE.

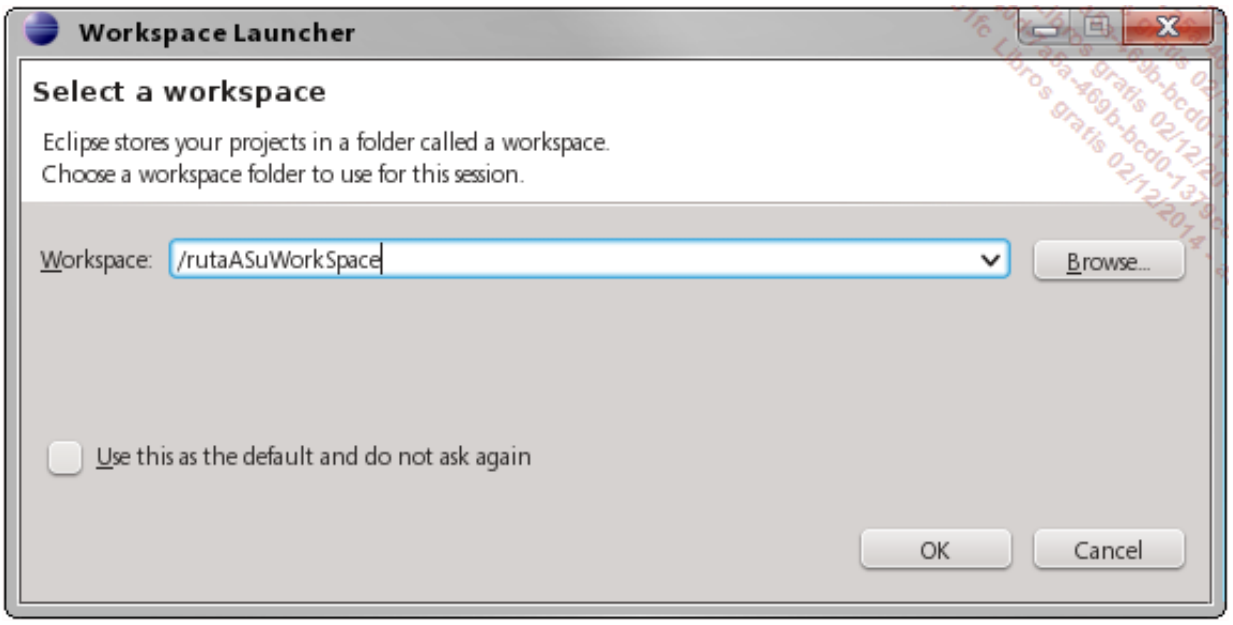
2. Instalación

- Para instalar Eclipse, diríjase a www.eclipse.org/downloads/ y descargue la versión de Eclipse para Java (a lo largo del libro se utilizará la versión Indigo). Cuando se haya descargado el archivo, extraiga la carpeta del archivo comprimido zip en una ubicación de su elección.

3. Configuración

- Inicie Eclipse mediante el acceso que se encuentra en la raíz de la carpeta extraída.

En la primera ejecución de Eclipse, deberá configurar su workspace. Se trata de su espacio de trabajo, todos los proyectos que desarrollará se crearán y se guardarán en esta carpeta.



Si se encuentra detrás de un proxy, debe configurarlo en las preferencias de Eclipse.

Configurar Eclipse en español

Si lo desea, puede configurar Eclipse en español siguiendo los pasos descritos a continuación:

- Para empezar, haga clic en **Help - Install New Software**. A continuación, haga clic en el botón**Add**. Introduzca el nombre que desee, la siguiente

URL:http://www.eclipse.org/babel/downloads.php y haga clic en **OK**. Para finalizar, seleccione todos los plugins disponibles e inicie la instalación.

Puede obtener más información en la dirección siguiente:http://www.eclipse.org/babel/downloads.php

Plugin ADT

1. Presentación

El siguiente paso consiste en la integración del plugin ADT (*Android Development Tools*) en Eclipse. Este plugin le da acceso a un entorno de desarrollo integrado para Android en Eclipse.

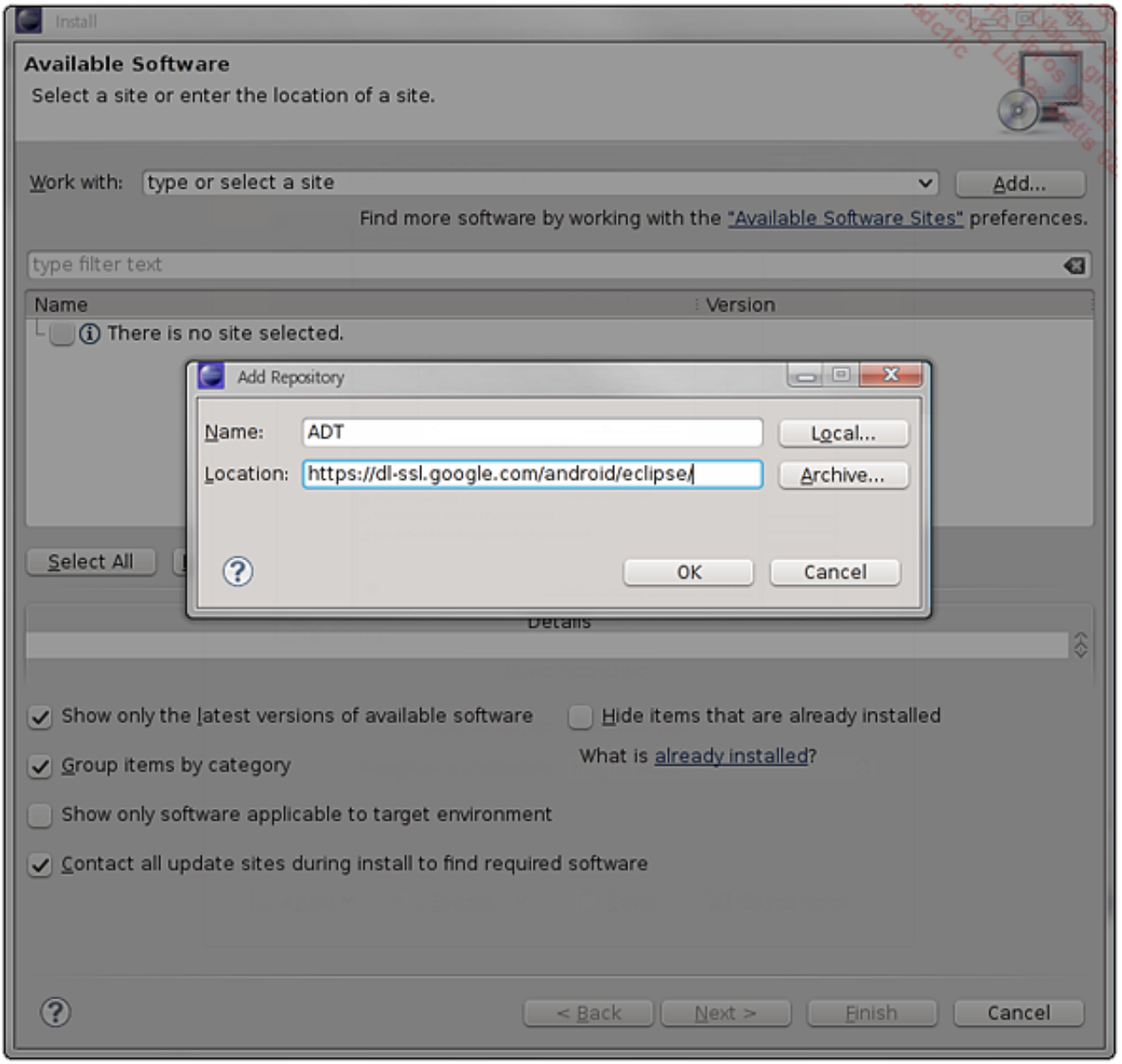
Utiliza las características de Eclipse para ayudarle en varias tareas:

- Crear fácilmente proyectos Android.
- Crear interfaces y componentes basados en el framework Android.
- Gestionar emuladores.
- Gestionar y visualizar logs.
- Depurar su aplicación utilizando el SDK Android.
- Exportar sus aplicaciones.

Una nueva versión del plugin ADT está disponible aproximadamente cada 3 o 4 meses. Recuerde comprobar las actualizaciones.

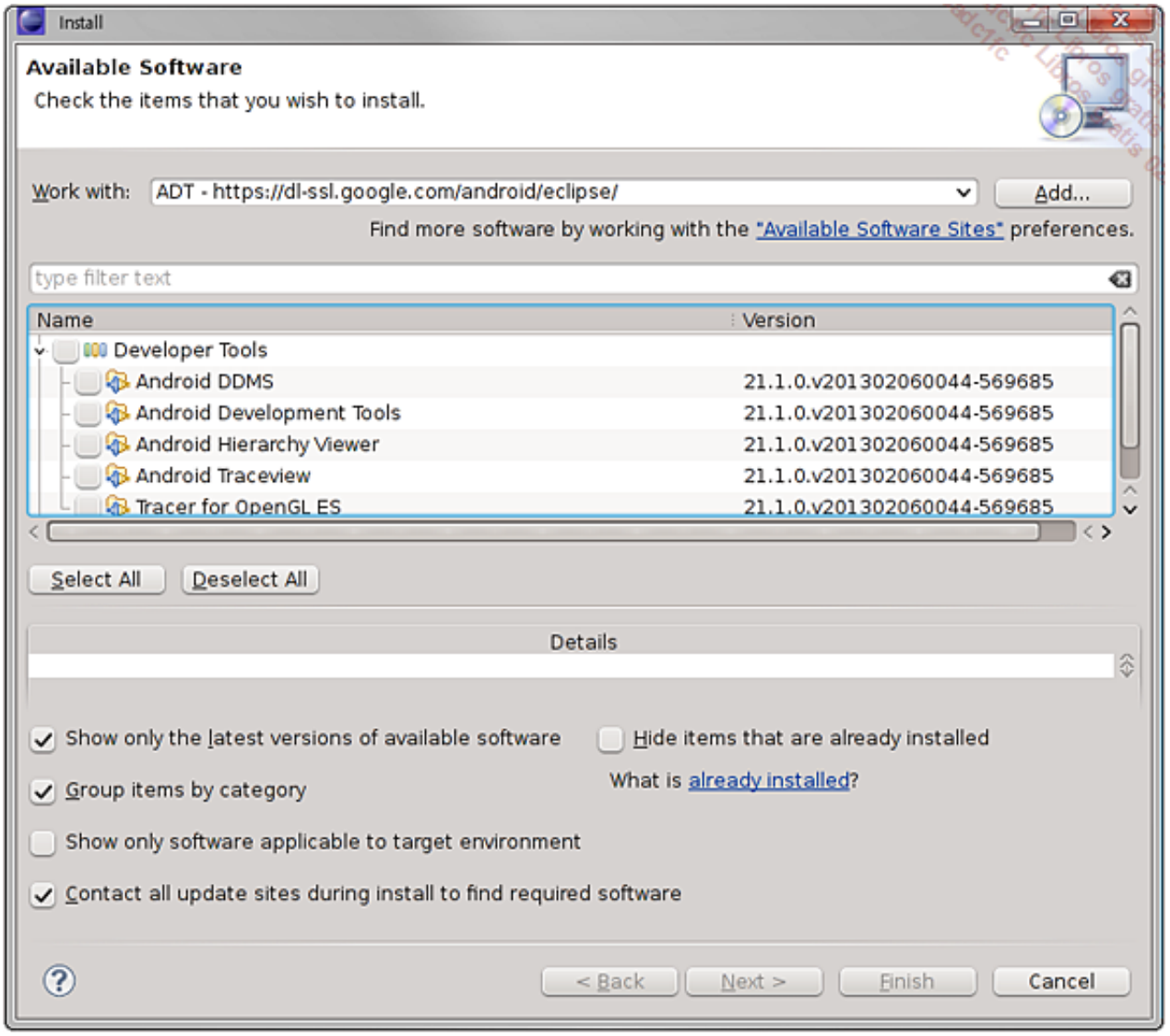
2. Instalación

Para comenzar, haga clic en **Help - Install new software**. A continuación, haga clic en el botón **Add**. Introduzca un nombre, la siguiente URL: https://dl-ssl.google.com/android/eclipse/ y haga clic en**OK**.



Si no accede a los recursos, reemplace el protocolo en la URL (https por un simple http).

Pasados unos segundos, aparecerá la siguiente ventana:



A continuación se enumeran los componentes que se instalarán:

- **Android DDMS** (*Android Dalvik Debug Monitor Server*): herramienta muy útil que integra la depuración y el control del teléfono o del emulador Android desde Eclipse.
- **Android Development Tools**: herramienta útil para el desarrollo de aplicaciones en Android.
- **Android Hierarchy Viewer**: herramienta que permite visualizar la arquitectura de sus vistas mediante un gráfico. Útil para la optimización de las vistas (véase el capítulo Creación de interfaces avanzadas - Optimizar sus interfaces).
- **Android Traceview**: permite mostrar el registro y los mensajes provenientes de su teléfono o emulador mediante un gráfico.

Seleccione todos los elementos y, a continuación, haga clic en **Next**. Compruebe su selección mediante el botón **Next**. Acepte las condiciones y los términos de uso y, finalmente, haga clic en**Finish**.

Cuando haya finalizado la instalación, reinicie Eclipse.

Si encuentra problemas realizando la instalación, puede descargar el archivo con la última versión del plugin ADT desde la siguiente dirección: http://developer.android.com/sdk/eclipse-adt.html y después importarlo desde Eclipse.

3. Configuración

Cuando Eclipse reinicia, aparece una nueva ventana para configurar su SDK.

Elija la opción **Use existing SDK** que le permite configurar el plugin para utilizar el SDK descargado en el apartado anterior. Haga clic en **Next** y, a continuación, en **Finish**.

Aparecerán tres nuevos iconos en la barra de herramientas de Eclipse:

- El primer icono sirve para iniciar el SDK Manager, actualizarlo o descargar nuevas versiones del SDK.
- El segundo icono se utiliza para gestionar los emuladores. De este modo, podrá crear fácilmente nuevos emuladores o modificar los existentes.
- El tercer icono sirve para escanear sus proyectos en busca de errores mediante la herramienta Android Lint (véase el capítulo Depuración y gestión de errores - Android Lint).

Emulador

1. Presentación

El emulador es una herramienta que permite simular un teléfono o una tablet Android. Por lo tanto, le permite desarrollar en Android incluso si no dispone de un teléfono o de una tablet Android.

También permite crear múltiples tipos de emulador para simular diferentes tipos de teléfonos, de tamaños de pantalla y de resoluciones.

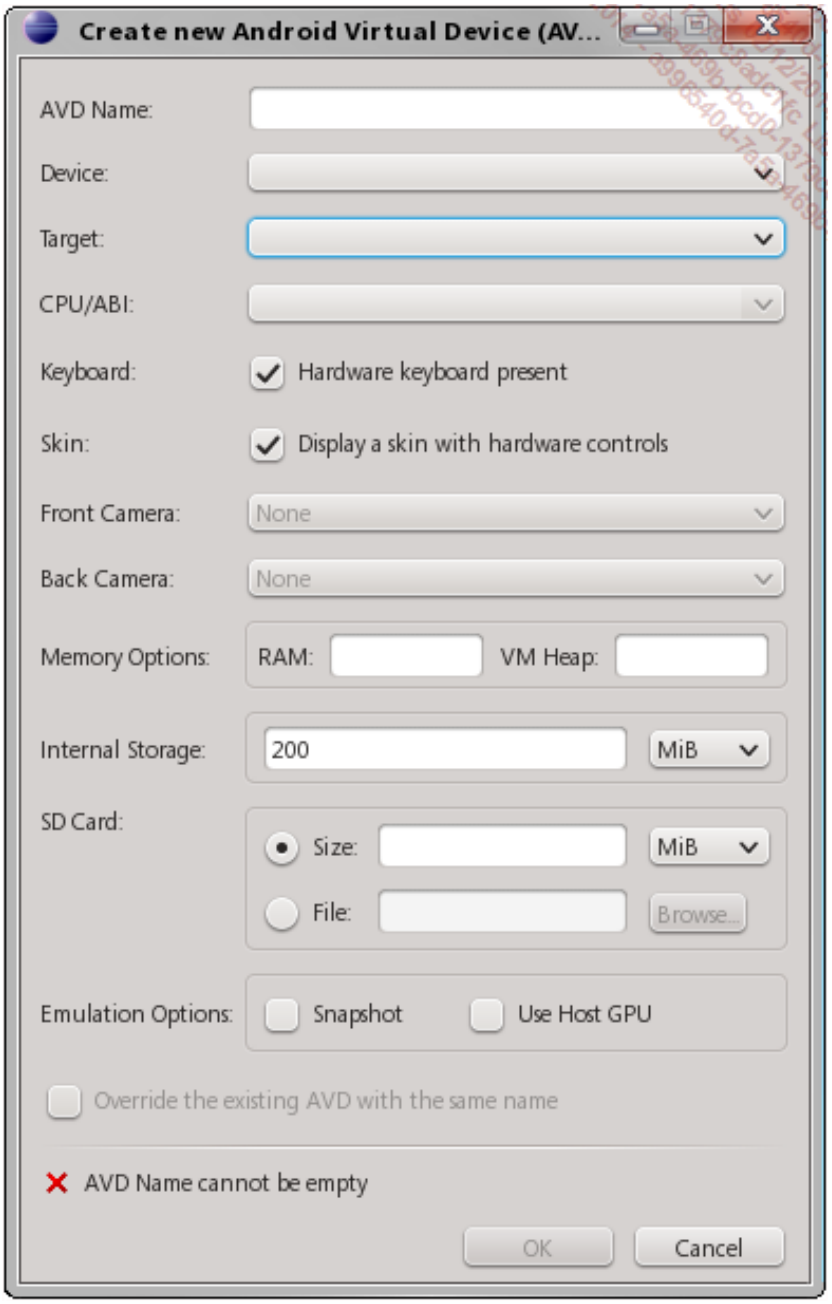
2. Creación

Como hemos visto anteriormente, Eclipse dispone de un icono dedicado a los emuladores.

Cuando haga clic en este icono, aparecerá una pantalla que le permitirá:

- Ver la lista de emuladores ya creados.
- Crear un nuevo emulador.
- Modificar un emulador.
- Eliminar o reparar un emulador.
- Ver los detalles de un emulador.
- Iniciar un emulador.

Haga clic en el botón **New** para crear un nuevo emulador.



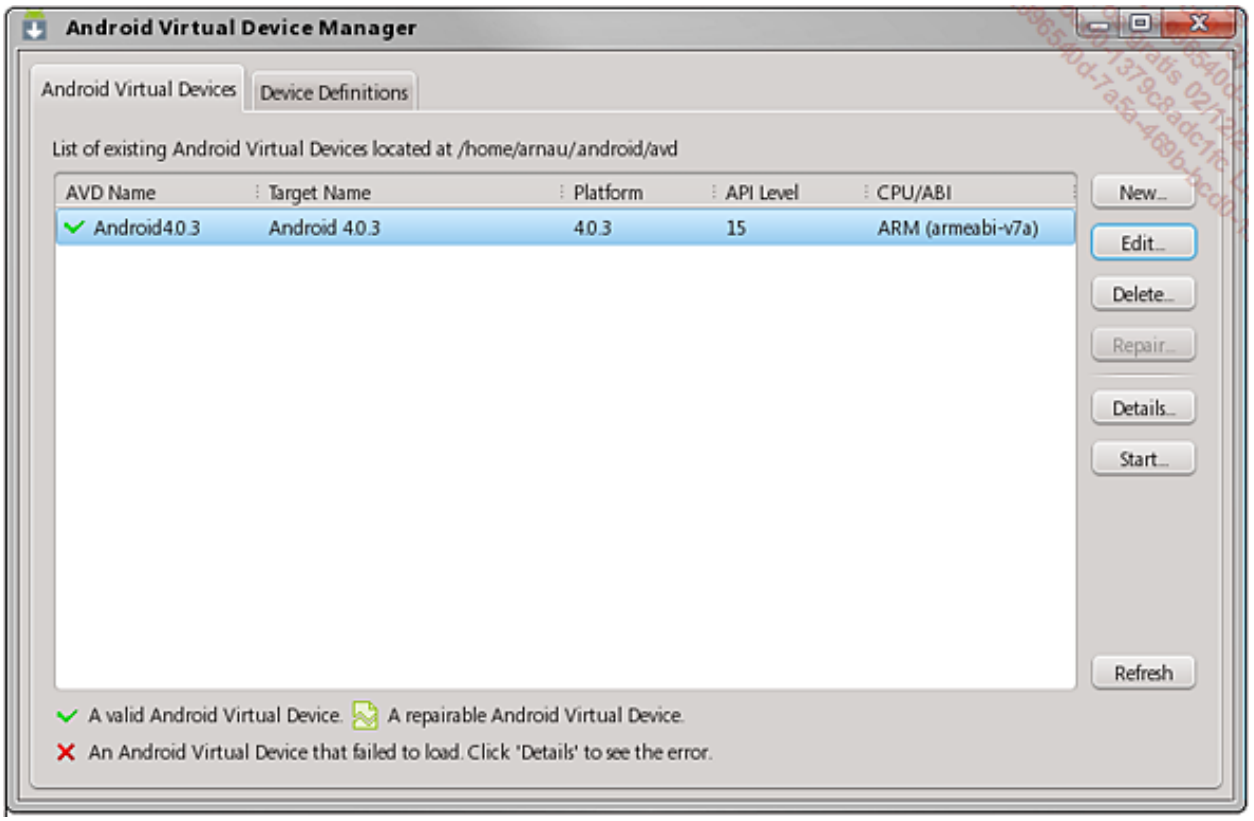
Puede especificar la siguiente información relativa a su emulador:

- **AVD Name**: nombre del emulador.
- **Device**: dispositivo predefinido con una serie de características hardware (GPS, acelerómetro, etc.). Puede definir sus propios dispositivos, mediante la pestaña **Device Definitions**, o usar los ya existentes por defecto.
- **Target**: versión de Android del emulador.
- **CPU/ABI**: procesador asociado al dispositivo.
- **Keyboard**: si desea emular un teclado conectado al dispositivo.
- **Skin**: si desea que se muestren los controles hardware en la emulación.
- **Front Camera**: seleccione el tipo de cámara frontal (ninguna, emulada, webcam).
- **Back Camera**: seleccione el tipo de cámara trasera (ninguna, emulada, webcam).
- **Memory Options**: definición de la memoria **RAM** y de la memoria dinámica de la VM.
- **Internal storage**: memoria interna del dispositivo.
- **SD Card**: tamaño de la tarjeta SD (opcional).

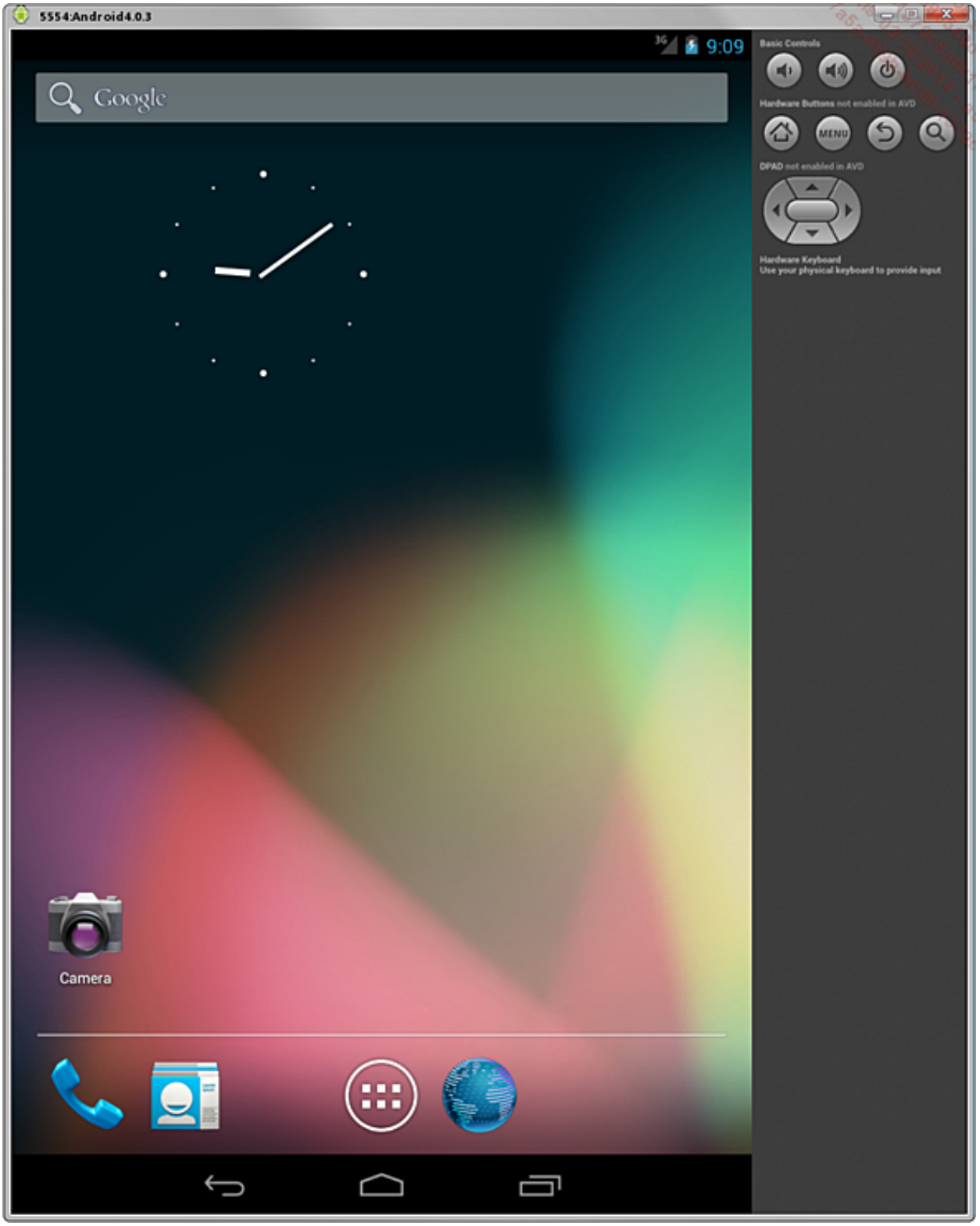
- **Emulation Options**: activar o desactivar tanto la posibilidad de guardar el estado del emulador (como para una máquina virtual, opcional) como la aceleración de la GPU mediante OpenGLES (opcional).

Ahora, podrá crear su primer emulador en Android 4.0.3.

Puede habilitar la aceleración de hardware en su emulador para que seá más reactivo (opción**GPU Emulation** en la sección **Hardware** del emulador).



Seleccione el emulador y haga clic en **Start** para iniciarlo. Tenga un poco de paciencia durante el transcurso de su arranque.

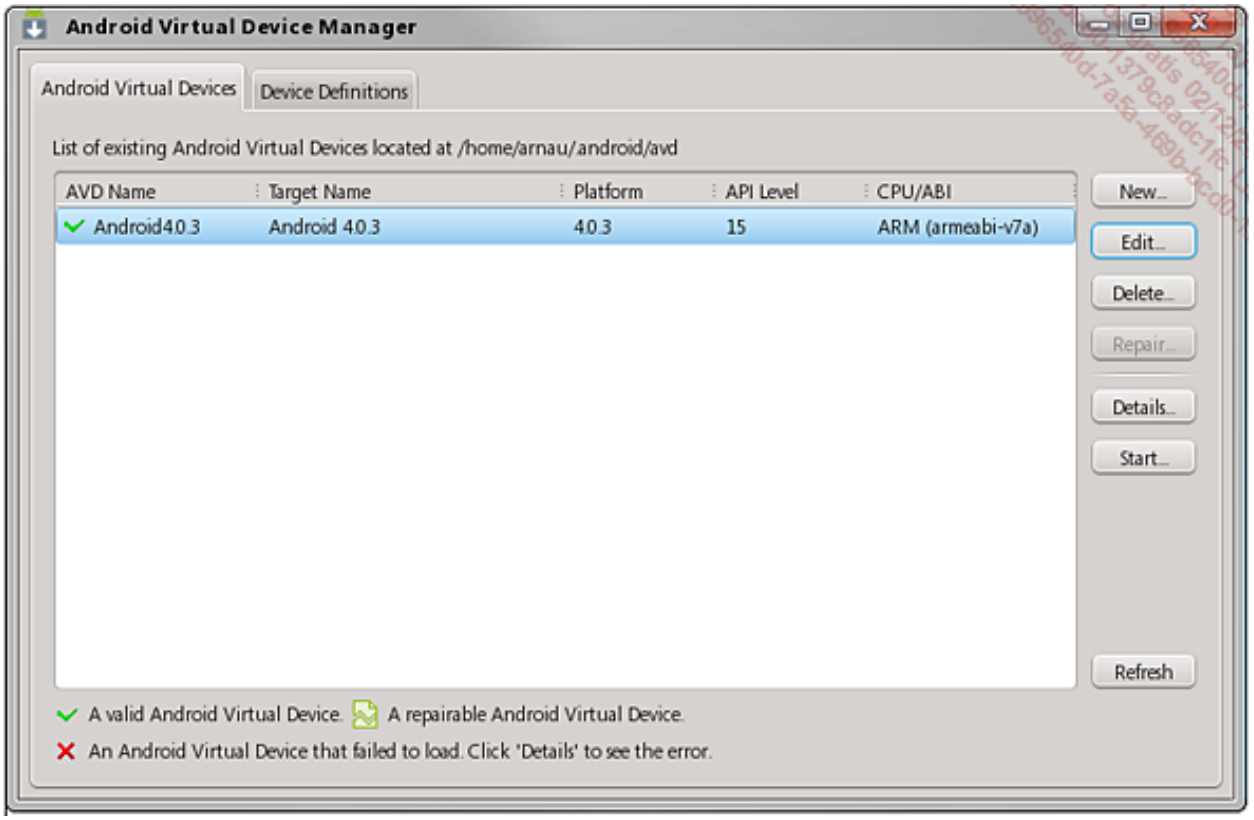


Una vez se haya iniciado el emulador, no es necesario cerrarlo y volverlo a iniciar cada vez que modifique su aplicación. Basta, simplemente, con ejecutarla de nuevo en el emulador deseado.

3. Configuración

Siempre puede modificar un emulador.

- Para ello, haga clic sobre el icono de gestión de emuladores en Eclipse, seleccione un emulador y haga clic en el botón **Edit**.



Todas las características del emulador se pueden modificar.

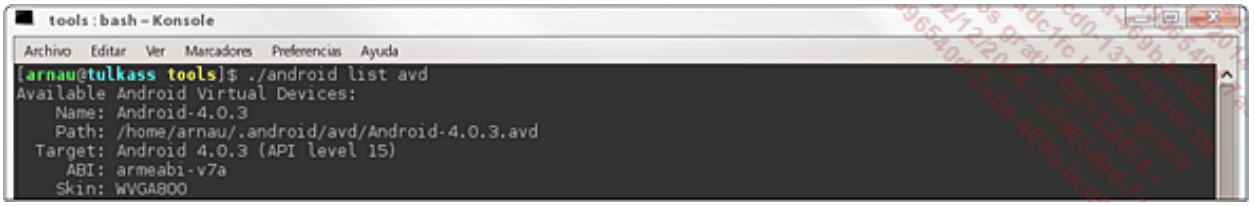
4. Creación de un emulador por línea de comandos

Puede enumerar el conjunto de emuladores creados mediante el siguiente comando:

```
./android list avd
```

Para ejecutar este comando debe ubicarse en la carpeta **tools** de su SDK.

Con el que obtendrá, por ejemplo:



También puede crear emuladores mediante el binario **Android** presente en la carpeta **tools** del SDK.

A continuación se muestra el ejemplo de un emulador cuyo nombre es **Android-4.0.3** y que funcionará con la versión 4.0.3 de Android.

```
./android create avd --target "android-15" --name "Android-4.0.3"
```

Obtendrá el siguiente resultado:

```
Auto-selecting single ABI armeabi-v7a
Android 4.0.3 is a basic Android platform.
Do you wish to create a custom hardware profile [no]
Created AVD 'Android-4.0.3' based on Android 4.0.3, ARM (armeabi-v7a) processor,
with the following hardware config:
hw.lcd.density=240
vm.heapSize=48
hw.ramSize=512
```

Puede responder ”yes” a la pregunta **Do you wish to create a custom hardware profile?** (¿Desea crear una configuración hardware específica?), para especificar los componentes hardware del emulador que está creando.

Principios generales

La elección de la arquitectura Android se basa en la idea de controlar los recursos y el consumo. Las aplicaciones Android se ejecutan en un sistema con restricciones (velocidad de procesador, memoria disponible, consumo de batería, diferencias en la visualización...).

Como desarrollador, deberá prestar especial atención a los siguientes puntos:

- La creación de nuevos objetos.
- El uso de recursos (procesador, RAM, almacenamiento, etc.).
- El consumo de la batería.
- La diversidad de tamaños y resoluciones de pantalla y de configuraciones de hardware.

Como cualquier página web, una pantalla Android sólo muestra una vista a la vez. La sucesión de páginas se almacena en una pila llamada **Back Stack** (véase el capítulo Principios de programación - Ciclo de vida de una actividad), lo que permite volver a la página anterior cuando se usa el botón atrás del teléfono (parecido al botón atrás de un navegador).

1. Dalvik

Android se basa en una máquina virtual particular llamada **Dalvik**, que tiene varias características específicas:

- El uso de registros y no de pilas.
- Una capa de abstracción entre la capa lógica y la capa de hardware.
- 30% de instrucciones menos que una JVM (*Java Virtual Machine*) clásica.
- Un tiempo de ejecución dos veces más rápido que una JVM clásica.
- Por defecto, las aplicaciones se aíslan unas de otras (1 aplicación = 1 proceso). Esto impide el efecto dominó ante la caída de una aplicación.

2. Arquitectura Android

La arquitectura Android se compone de cinco partes diferenciadas:

- **Aplicación**: representa el conjunto de de aplicaciones proporcionadas con Android (e-mail, SMS, calendario, etc.).
- **Framework Android**: representa el framework que permite a los desarrolladores crear aplicaciones accediendo al conjunto de APIs y funcionalidades disponibles en el teléfono (fuentes de contenido, gestor de recursos, gestor de notificaciones, gestor de actividades, etc.).
- **Librerías**: Android dispone de un conjunto de librerías que utilizan los distintos componentes del sistema.
- **Android Runtime**: contiene, entre otros, la JVM Dalvik.
- **Linux Kernel**: el núcleo Linux (2.6) que proporciona una interfaz con el hardware, gestionando la memoria, los recursos y los procesos Android.

3. NDK (Native Development Kit)

Es posible desarrollar de forma nativa en Android utilizando el NDK (*Native Development Kit* - Kit de desarrollo nativo), que se basa en C/C++.

También puede utilizar un mecanismo Java que le permitirá llamar a código nativo desde una clase JNI (*Java Native Interface*).

El NDK está disponible en la siguiente dirección:http://developer.android.com/sdk/ndk/index.html

El NDK le da acceso a:

- Un conjunto de herramientas que permiten generar código nativo desde archivos fuente C/C++.
- Una herramienta de creación de archivos apk (véase la siguiente sección: APK (Android Package)) a partir de código nativo.
- La documentación, con ejemplos y tutoriales.

4. APK (Android Package)

Un apk es el archivo binario que representa una aplicación. Este formato se utiliza para distribuir e instalar aplicaciones.

Para crear un apk, se debe compilar y empaquetar en un archivo una aplicación Android. Este archivo contendrá:

- El código de la aplicación compilada (.dex).
- Los recursos.
- Los assets.
- Los certificados.
- El archivo de manifiesto.

a. Exportación y certificación de una aplicación

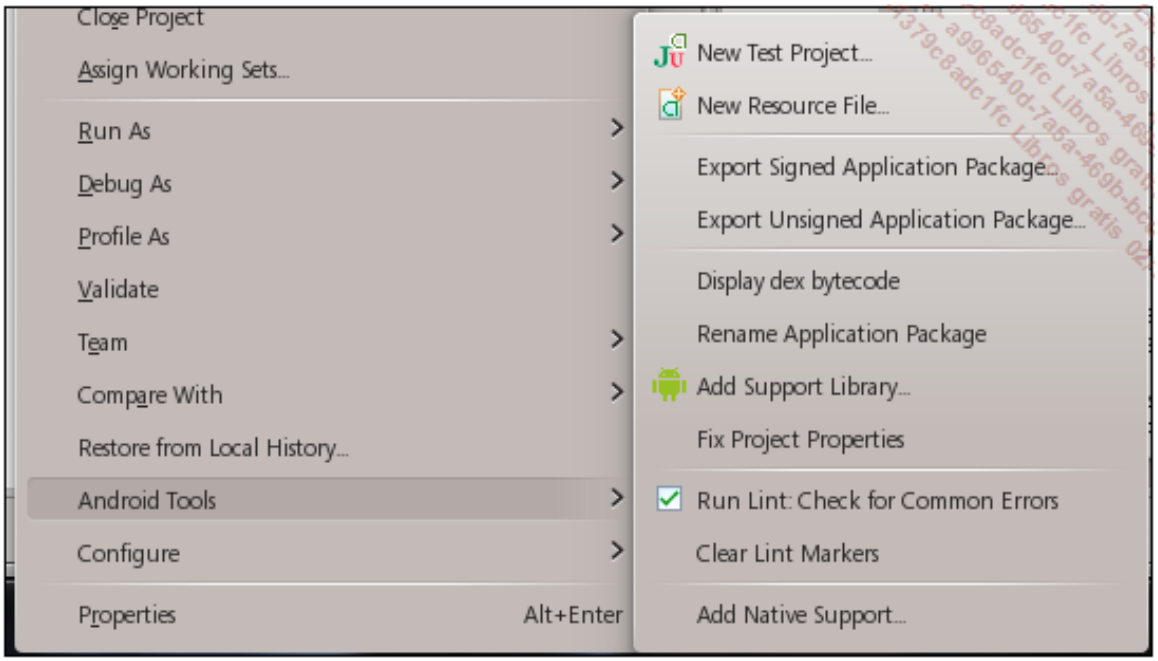
Para poder publicar su aplicación en el Market (**Google Play**), necesita crear un archivo apk.

Para generar su apk desde **Eclipse**, Android proporciona una herramienta que se llama **aapt**, incluida en el SDK Android e integrada en Eclipse mediante el plugin ADT.

Puede generar dos tipos de apk:

- Un apk firmado.
- Un apk sin firmar: este formato puede usarse para probar su aplicación pero no puede publicarse en el Market.

Para generar un apk, haga clic con el botón derecho sobre el proyecto deseado y, en el menú, haga clic en **Android Tools**.



Puede observar las dos opciones:

- **Export Signed Application Package** (Exportar un APK firmado).
- **Export Unsigned Application Package** (Exportar un APK sin firmar).

Vamos a seleccionar la opción **Export Signed Application Package**.

Confirme el proyecto del que desea generar el apk mediante el botón **Next**.

Para generar un archivo apk firmado, necesitará un **keystore** (almacén de claves), que sirve para albergar las claves utilizadas para firmar sus aplicaciones. Puede o bien usar un keystore que ya usaba previamente, o bien crear uno nuevo.

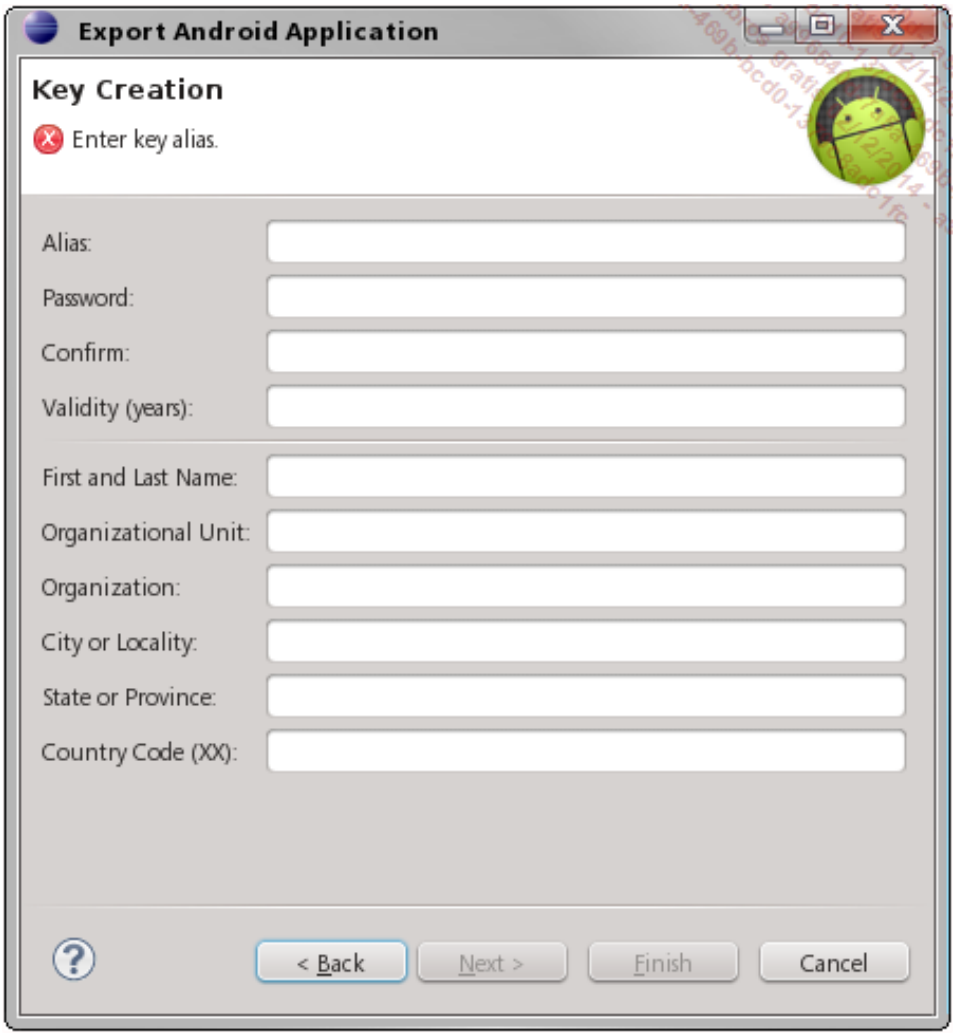
Para crear un keystore hay que:

- Elegir una ubicación para guardar el keystore.
- Indicar una contraseña para el uso del keystore.

Un keystore se compone de varias claves de firma. Cada clave puede servir, por ejemplo, para firmar una aplicación distinta.

La siguiente pantalla le permite introducir los datos siguientes sobre su clave:

- Alias: Identificador de la clave.
- Contraseña y confirmación.
- Duración de validez, en años.
- Introducir la información del resto de campos (nombre y apellidos, población, país, etc.).



El último paso consiste en elegir el nombre y la ubicación del apk generado.

Debe conservar su clave de certificación cuando haya publicado su aplicación en el Market. Si la pierde, no podrá actualizar su aplicación nunca más.

También puede crear una clave de firma por línea de comandos mediante el ejecutable **keytool**.

```
[arnau@tulkass ~]$ keytool -genkey -keystore claves -alias "clave firma android" -keyalg RSA -validity 36500
Introduzca la contraseña del almacén de claves:
Volver a escribir la contraseña nueva:
¿Cuáles son su nombre y su apellido?
  [Unknown]:  Arnau
¿Cuál es el nombre de su unidad de organización?
  [Unknown]:  www.android.es
¿Cuál es el nombre de su organización?
  [Unknown]:
¿Cuál es el nombre de su ciudad o localidad?
  [Unknown]:
¿Cuál es el nombre de su estado o provincia?
  [Unknown]:
¿Cuál es el código de país de dos letras de la unidad?
  [Unknown]:  ES
¿Es correcto CN=Arnau, OU=www.android.es, O=Unknown, L=Unknown, ST=Unknown, C=ES?
  [no]:  sí

Introduzca la contraseña de clave para <clave firma android>
        (INTRO si es la misma contraseña que la del almacén de claves):
Volver a escribir la contraseña nueva:
```

En Windows, el ejecutable keytool se encuentra en la carpeta bin de la carpeta de instalación de Java.

También puede firmar su aplicación a posteriori mediante el ejecutable **jarsigner**.

```
jarsigner -verbose -verify -certs miAplicacion.apk
```

Componentes Android

El framework Android se compone de algunos elementos esenciales para la creación de aplicaciones. En esta sección, se explicarán estos distintos componentes.

1. Activity (Actividad)

Una actividad es el componente principal de una aplicación Android. Representa la implementación y las interacciones de sus interfaces.

Tomemos, por ejemplo, una aplicación que enumere todos los archivos mp3 que se encuentran en su teléfono. El proyecto podría descomponerse de la siguiente manera:

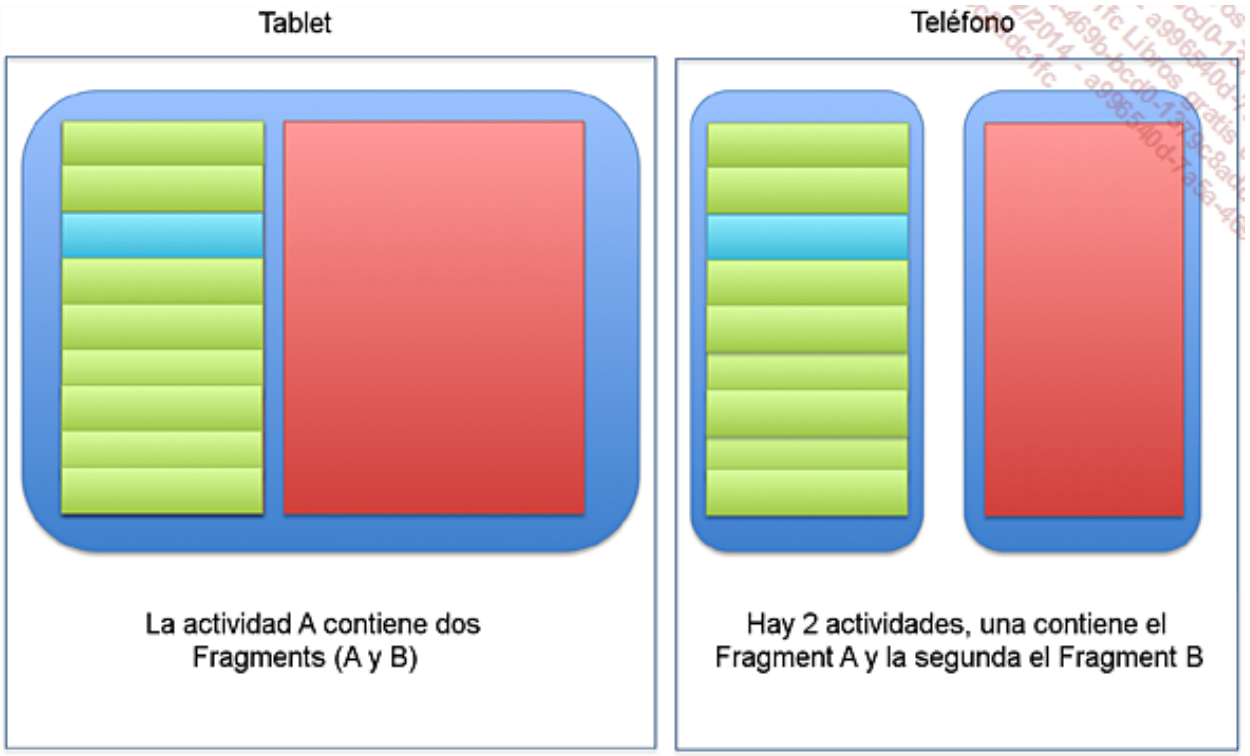
- Una vista para mostrar la lista de archivos mp3.
- Una actividad para implementar la interfaz y las interacciones del usuario con la lista de archivos mp3.

El framework Android puede decidir "matar" una actividad para liberar recursos para otras aplicaciones (sólo si su actividad ya no está en primer plano) (véase la sección Ciclo de vida de una actividad, más adelante en este capítulo).

2. Fragment (Fragmento)

Este concepto se introdujo a partir de la versión 3.0 de Android. Permite construir interfaces más flexibles (smartphone/tablet), dinámicas y fáciles de utilizar.

Un fragmento puede considerarse como una parte de una interfaz. Por lo tanto, una interfaz (actividad) puede componerse de varios fragmentos.



Tomemos una interfaz compuesta de dos vistas. En vez de implementar dos actividades distintas, se podrían utilizar los fragmentos. Usando esta alternativa, se obtendrían dos interfaces diferenciadas para teléfono o para tablet.

- Para un teléfono: los fragmentos se separan en dos actividades diferentes. La selección de

un elemento de la actividad A (el fragmento A se incluye en la actividad A) mostrará la actividad B (el fragmento B se incluye en la actividad B).

- Para una tablet: su interfaz se compondrá de una sola actividad, pero ésta estará separada en dos fragmentos. Cuando el usuario pulse un elemento del fragmento A, actualizará todo el contenido del fragmento B sin iniciar una actividad nueva.

3. Servicio (Service)

Un servicio, a diferencia de una actividad, no tiene interfaz pero permite la ejecución de un tratamiento en segundo plano (una operación larga o una llamada remota). Un servicio no se detendrá mientras que no se interrumpa o termine.

Por ejemplo, se puede utilizar un servicio para escuchar música en segundo plano en su teléfono.

4. Broadcast receiver (Receptor de eventos)

Un Broadcast receiver es un componente que reacciona con un evento de sistema (véase la sección Intent (Intención)). Permite, por ejemplo, saber cuándo:

- El teléfono recibe un SMS.
- El teléfono se enciende.
- La pantalla está bloqueada, etc.

Los broadcast receivers no tienen interfaz de usuario y deben realizar tareas ligeras. Si necesita ejecutar una tarea pesada en la recepción de un Broadcast receiver, puede, por ejemplo, iniciar un servicio.

Un Broadcast receiver no puede realizar modificaciones en sus interfaces, sólo puede mostrar una notificación, iniciar una actividad o un servicio, etc.

5. Content provider (Proveedor de contenido)

Un content provider permite compartir los datos de una aplicación. Estos datos pueden estar almacenados en una base de datos SQLite (véase el capítulo Persistencia de datos - Almacenamiento en base de datos), en archivos o en la web. El objetivo es permitir a las aplicaciones consultar estos datos.

El sistema Android ofrece content providers (disponibles por defecto) que pueden usarse en sus aplicaciones:

- Contactos.
- Agenda.
- Multimedia, etc.

6. Intent (Intención)

Los componentes Android (Actividad, Servicio y Broadcast receiver) se comunican mediante mensajes de sistema que se denominan intents. Los emite el terminal para avisar a la cada aplicación de la ocurrencia de eventos.

También puede crear sus propios intents para, por ejemplo, iniciar otras actividades (véase el capítulo Comunicación entre vistas - aplicaciones).

Hay dos tipos de intents:

- **Explícito**: como, por ejemplo, llamar a una actividad conocida para que ejecute una acción en concreto.
- **Implícito**: solicitar al sistema qué actividad puede ejecutar una acción específica (por ejemplo, abrir un archivo PDF).

a. Intent-filter (Filtro de intenciones)

Los filtros de intenciones sirven para indicar a una actividad, servicio o broadcast receiver qué intents pueden tratar de forma implícita. Con ello, todo componente que desee ser avisado por intenciones debe declarar filtros de intenciones.

La manera más común de declarar filtros de intenciones consiste en utilizar la etiqueta **intent-filter**en el manifiesto (AndroidManifest.xml) de su aplicación.

En la creación de un proyecto Android, la actividad principal del proyecto contiene filtros de intenciones.

```
<activity android:name=".HelloAndroidActivity"
          android:label="@string/app_name" >
  <intent-filter>
     <action android:name="android.intent.action.MAIN" />
     <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Estos dos filtros de intenciones significan, respectivamente, que la actividad:

- Es la actividad principal de la aplicación.
- Pertenece a la categoría Launcher, es decir, estará disponible desde el menú principal de inicio de aplicaciones de Android.


También puede crear filtros de intenciones dinámicamente gracias a la clase **IntentFilter**.

b. pendingIntent

Un pendingIntent corresponde a la descripción de un intent asociado a una acción. Se crea mediante uno de los métodos siguientes:

- **getActivity(Context, int, Intent, int)**: permite generar un pendingIntent que apunte a una actividad.
- **getBroadcast(Context, int, Intent, int)**: permite generar un pendingIntent que apunte a un Broadcast Receiver.
- **getService(Context, int, Intent, int)**: permite generar un pendingIntent que ejecute un servicio.

Permite, a un mecanismo externo a su aplicación (NotificationManager, AlarmManager, etc.), tener los privilegios necesarios para desencadenar en un momento dado un intent en el interior de su aplicación.

El ejemplo más común es el pendingIntent que sirve para desencadenar una notificación tras la recepción de un mensaje.

La clase Application

Cada aplicación Android tiene una clase Application. Esta clase permanece instanciada a lo largo de todo el ciclo de vida de su aplicación. Puede sobrecargar esta clase para:

- Reaccionar ante un Broadcast Receiver.
- Transferir objetos entre los componentes de una aplicación.
- Gestionar los recursos que utiliza su aplicación.
- Declarar constantes globales para su aplicación.

Una vez se haya sobrescrito esta clase, debe declararla en su archivo de manifiesto para que se tome en consideración.

```
<application android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:name=".MyApplication">
```

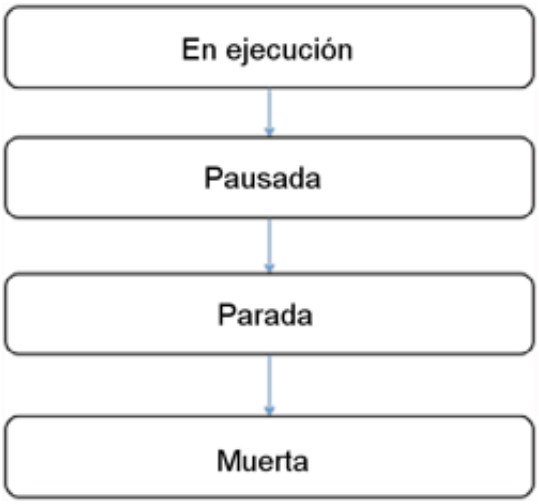
La clase Application se instanciará en el arranque de la aplicación.

Ciclo de vida de una actividad

Por defecto, cada aplicación Android se ejecuta en un proceso separado. Android gestiona los recursos disponibles en el dispositivo y puede, si fuera necesario, cerrar aplicaciones para liberar recursos (excepto aplicaciones en ejecución).

La elección de la aplicación que se cerrará depende fuertemente del estado del proceso en el que se encuentra. Si Android debe elegir entre dos aplicaciones que se encuentran en el mismo estado, elegirá la que se encuentre en este estado desde hace más tiempo.

1. Estado de una actividad



Una actividad puede encontrarse en cuatro estados distintos:

- **En ejecución**: la actividad se encuentra en primer plano y recibe las interacciones del usuario. Si el dispositivo necesita recursos, se matará la actividad que se encuentra en el fondo del back stack.
- **Pausada**: la actividad está visible pero el usuario no puede interactuar con ella (oculta por un cuadro de diálogo, por ejemplo). La única diferencia con el estado anterior es la no recepción de eventos de usuario.
- **Parada**: la actividad ya no es visible pero sigue en ejecución. Todos los datos relativos a su ejecución se conservan en memoria. Cuando una actividad pasa a estar parada, debe guardar los datos importantes y detener todos los tratamientos en ejecución.
- **Muerta**: se ha matado la actividad, ya no está en ejecución y desaparece de la **back stack**. Todos los datos presentes en caché que no se hayan guardado se pierden.

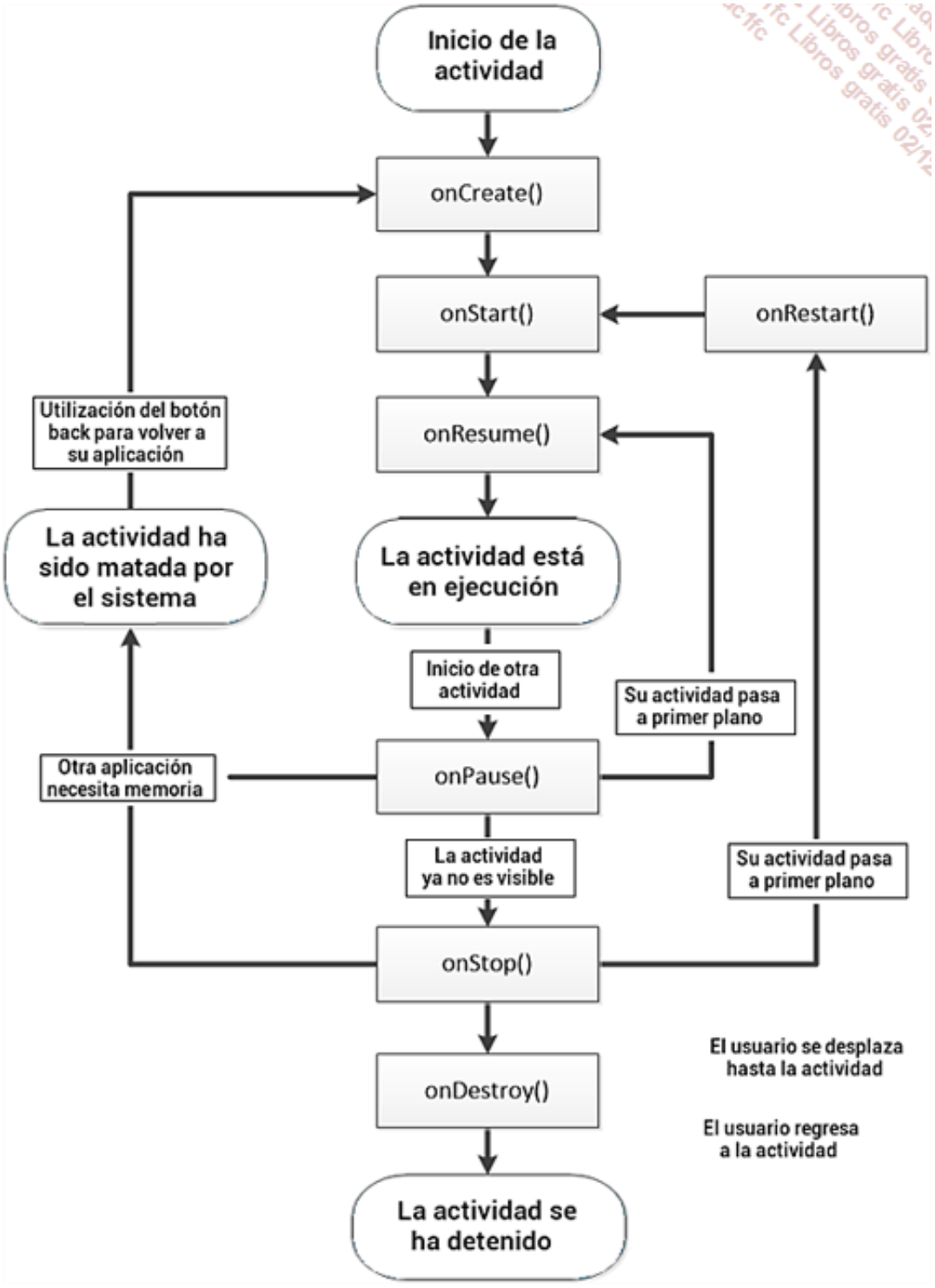
2. Back stack

Todas las actividades iniciadas se almacenan en una lista que, generalmente, se denomina **back stack**. Cuando se inicia una actividad nueva, ésta se añade en la cima de la back stack.

Si el usuario pulsa el botón ”atrás” de su dispositivo, la actividad en curso se cierra y la que se encontraba inmediatamente debajo en la back stack se abre.

3. Ciclo de vida

El ciclo de vida de una actividad es bastante complejo y su comprensión es indispensable en el desarrollo para Android. El siguiente esquema resume este ciclo de vida:



Cuando se inicia la actividad, se invoca al método **onCreate**. En este método, debe inicializar su actividad, cargar su interfaz y arrancar sus servicios.

A esta llamada le sigue el método **onStart**, que permite indicar el inicio efectivo de la aplicación (ahora ya es visible).

A continuación, se invoca al método **onResume** para ejecutar todos los tratamientos necesarios para el funcionamiento de la actividad (thread, proceso, tratamiento), inicializar variables y listeners. Estos tratamientos deberán detenerse cuando se invoque al método **onPause** y relanzarse, si fuera necesario, cuando se produzca una futura llamada al método **onResume**.

Después de estas tres llamadas, la actividad se considera utilizable y puede recibir interacciones de los usuarios.

Si otra actividad pasa a primer plano, la actividad en ejecución pasará a estar pausada. Justo antes de la llamada al método **onPause**, se invocará al método **onSaveInstanceState** para permitirle guardar los datos importantes de la actividad. Estos datos podrán aplicarse a futuras ejecuciones

de la actividad con la llamada al método **onRestoreInstanceState**.

Por ejemplo, en una aplicación de gestión de mensajes, el método **onSaveInstanceState** podría servir para guardar el borrador de un SMS que se estaba escribiendo en ese mismo momento. Esto permitirá al usuario volverse a encontrar el borrador al volver a la aplicación.

El método **onPause** permite detener los tratamientos realizados (tratamiento no necesario si la actividad no es visible) por la actividad (tratamiento, thread, proceso).

Si su actividad pasa a estar visible de nuevo, se realizará una llamada al método **onResume**.

El paso de la actividad al estado ”parada” tiene asociado una llamada al método **onStop**. En este método hay que detener todos los tratamientos restantes.

Una vez parada, su actividad puede:

- **Volver a iniciarse**: acción realizada por una llamada al método **onStart** siguiendo el ciclo de vida normal de la actividad.
- **Terminar**: se realiza con una llamada al método **onDestroy**, en el que deberá parar todos los tratamientos restantes, cerrar todas las conexiones a la base de datos, todos los archivos abiertos, etc. Puede provocar la llamada al método **onDestroy** utilizando el método **finish**.


Es posible sobrecargar todos los métodos del ciclo de vida de una actividad.

Contexto de una aplicación

El contexto representa el estado actual de una aplicación y la información relativa a su entorno. Sirve para recuperar objetos enviados por otros elementos de su aplicación.

Puede obtener el contexto de cuatro formas distintas:

- **getApplicationContext()**: permite obtener el contexto de su aplicación.
- **getContext()**: permite obtener el contexto en el que se ejecuta la vista actual.
- **getBaseContext()**: permite obtener el contexto definido mediante el método**setBaseContext()**.
- **this** (sólo cuando se encuentre en una clase que herede de forma directa o indirecta de la clase **Context**).


La clase Activity extiende la clase Context.

Manifiesto

El archivo **AndroidManifest.xml**, que se encuentra en la raíz de todo proyecto Android, es lo más parecido a una descripción completa de la aplicación (componentes, actividades, servicios, permisos, proveedores de contenido, etc.).

Cada componente que requiera una declaración en el manifiesto se abordará en la sección dedicada a dicho componente o funcionalidad.

Cuando publica una aplicación en el Market, éste último se encarga de recorrer su archivo manifiesto para recopilar el máximo de información sobre su aplicación (nombre del package, icono, descripción, contenido, versión, etc.).

Vamos a describir las secciones generales e indispensables de un manifiesto. El comienzo de un archivo de manifiesto es, con algunos detalles, casi idéntico al de cualquier otra aplicación.

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.eni.android.hello"
    android:versionCode="1"
    android:versionName="1.0" >
...
</manifest>
```

Esta primera parte contiene:

- La declaración xmlns (obligatoria).
- El package (identificador) de su aplicación (elegido en la creación de su proyecto).
- El código de la versión actual de su aplicación. Este código representa un valor entero que se incrementará con cada nueva versión de su aplicación.
- El nombre de la versión (visible en el Market).

El contenido del manifiesto varía en función de la aplicación:

- **Las permissions** (véase la sección Permissions (permisos) en este capítulo).
- **Las instrumentations**: proporcionan un framework de pruebas para una aplicación.
- **uses-sdk**: este nodo permite especificar las versiones mínima (**minSDKVersion**) y máxima (**maxSDKVersion**) del SDK compatible con la aplicación, así como la versión utilizada para el desarrollo de la misma (**targetSDKVersion**).
- **uses-configuration**: este atributo permite detallar las especificidades de navegación compatibles con su aplicación, tales como los teclados físicos, trackball, lápiz, etc. Este nodo es opcional.
- **uses-feature**: este nodo permite indicar los elementos de hardware requeridos por la aplicación (audio, bluetooth, cámara, localización, NFC, etc.). Este nodo es opcional.
- **supports-screens**: este atributo permite especificar las dimensiones de pantalla soportadas por la aplicación (smallScreens, normalScreens, largeScreens, xlargeScreens).
- **La aplicación**: define la estructura y el contenido de la aplicación (actividad, servicio, etc.).

1. Instalación de la aplicación

Puede especificar las preferencias de instalación en un medio (instalación en el almacenamiento interno o externo) gracias al atributo **installLocation**. Por defecto, la instalación se realiza en el almacenamiento interno.

Si desea instalar la aplicación en el almacenamiento externo, este último quedará inutilizable si el almacenamiento externo se conecta a un ordenador, por ejemplo.

Algunas aplicaciones deben instalarse en el almacenamiento interno del dispositivo:

- Las aplicaciones que tengan un widget o un fondo de pantalla animado.
- Las aplicaciones que representan teclados, etc.

2. Descripción de su aplicación

En la sección application del manifiesto tendrá que describir todas las características específicas y todos los componentes de su aplicación:

- Nombre de la aplicación.
- Icono de la aplicación.
- Tema utilizado por la aplicación.
- Los distintos componentes de la aplicación (Actividad, Servicio, Proveedor de contenido, Receptor de eventos) así como las características específicas de cada componente (componente principal, presencia en el inicio de la aplicación, uso de datos).

Permissions (permisos)

Una de las características específicas de Android está relacionada con el sistema de permisos. Permite, a cada aplicación, definir los permisos que desea tener.

Cada *permission* se corresponde con el derecho de acceso a una funcionalidad o a un dato.

Por defecto, las aplicaciones no tienen ninguna *permission* y deben solicitar una *permission* cada vez que lo requieran. Por ejemplo, esto permite al usuario saber la lista de todas las *permissions* que solicita una aplicación antes de instalarla.

También puede crear *permissions* para que otras aplicaciones tengan el derecho de utilizar sus datos o sus servicios, etc.

Las *permissions* se declaran fuera de la etiqueta **application** de su manifiesto.

1. Utilizar una permission

Solicitar una *permission* es muy fácil: se añade la etiqueta **<uses-permission>** en el archivo**AndroidManifest.xml**.

Si desea utilizar una conexión a Internet en su aplicación, deberá por tanto utilizar la siguiente línea en su manifiesto:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Todas las *permissions* de sistema empiezan por **android.permission**, puede encontrarlas en esta dirección: http://developer.android.com/reference/android/Manifest.permission.html

Si, por ejemplo, su aplicación utiliza la conexión a Internet sin solicitar la *permission*, la excepción **SecurityException** le informará, en la mayoría de los casos, que le falta declarar la*permission* en su manifiesto.

2. Declarar sus permissions

También puede declarar sus propias *permissions*. La declaración de una *permission* se realiza en el archivo **AndroidManifest.xml** mediante la etiqueta **<permission>**.

Para declarar una *permission*, necesitará tres datos:

- **El nombre de su permission**: para evitar un conflicto de nombres, prefije su *permission* con el identificador del package de la aplicación.
- **Una etiqueta para su permission**: un texto corto y comprensible que indicará la descripción de la *permission*.
- **Una descripción de la permission**: un texto más completo para explicar con mayor claridad los datos accesibles mediante la *permission*.

A continuación se muestra un ejemplo:

```
<permission
android:name="com.eni.android.permission.MI_PERMISSION"
```

Y, a continuación, las cadenas de caracteres que sirven de etiqueta y de descripción para esta

nueva *permission*:

```
<resources>
    <string name = "label_permission"> Etiqueta de
        mi permission </string>
    <string name = "label_description"> Descripción
        de mi permission </string>
</resources>
```

También debe detectar las violaciones de seguridad en el uso de *permissions* que haya definido.

Si desea que el uso de un dato, actividad o servicio en particular de su aplicación, requiera la declaración de una *permission*, puede indicarlo en su archivo de manifiesto, en cualquier parte donde se exija la *permission* con el atributo **android:permission**.

Tomemos, por ejemplo, el caso de un servicio que obliga a declarar una *permission* a la aplicación que desea utilizarlo:

```
<service android:name="com.eni.android.MiServicio"
android:permission="com.eni.android.permission.MI_PERMISSION">
```

En este ejemplo, cualquier aplicación que desee utilizar el servicio "MiServicio" debe declarar, obligatoriamente, la *permission* "MI_PERMISSION".

Si su *permission* involucra a un proveedor de contenido (content provider), puede distingir la*permission* de lectura de la *permission* de escritura (**android:readPermission** y**android:writePermission**).

También puede comprobar el uso de su *permission* directamente en el código de su servicio, por ejemplo. Para ello, utilice el método **checkCallingPermission()** que devolverá, respectivamente, PERMISSION_GARANTED, si la *permission* se ha solicitado correctamente, o PERMISSION_DENIED, en caso contrario.


Todos los errores de *permission* aparecen en tiempo de ejecución, y no en la compilación.

Creación del proyecto

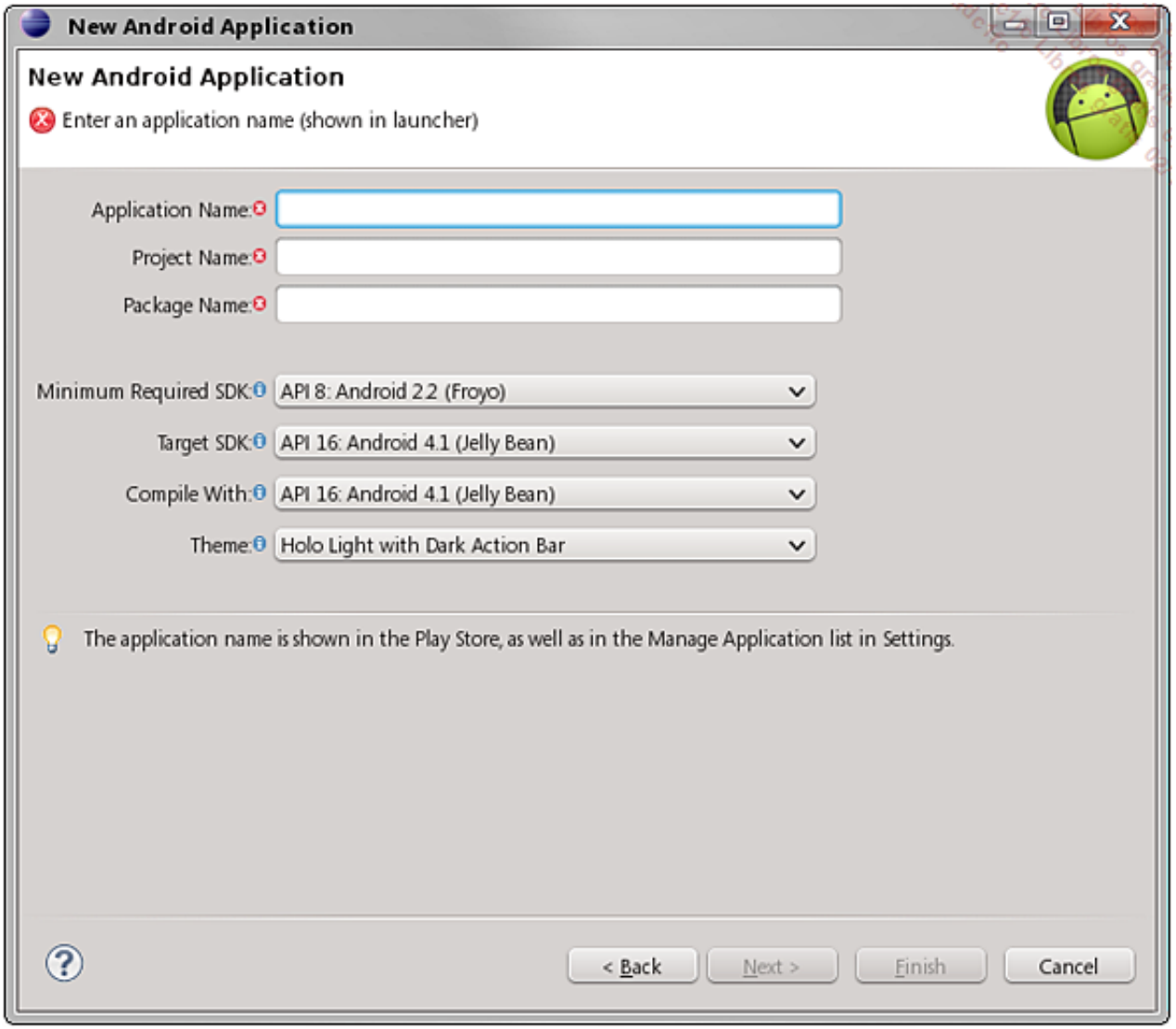
Crear un proyecto Android es sencillo.

- Haga clic en **File - New - Android Application Project**. Si la opción **Android Application Project**no apareciera, haga clic en **Other** y, en la sección Android, seleccione **Android Application Project** y haga clic en el botón **Next**.

Aparecerá un formulario que le servirá para crear su proyecto. Este formulario se compone de varias pantallas.

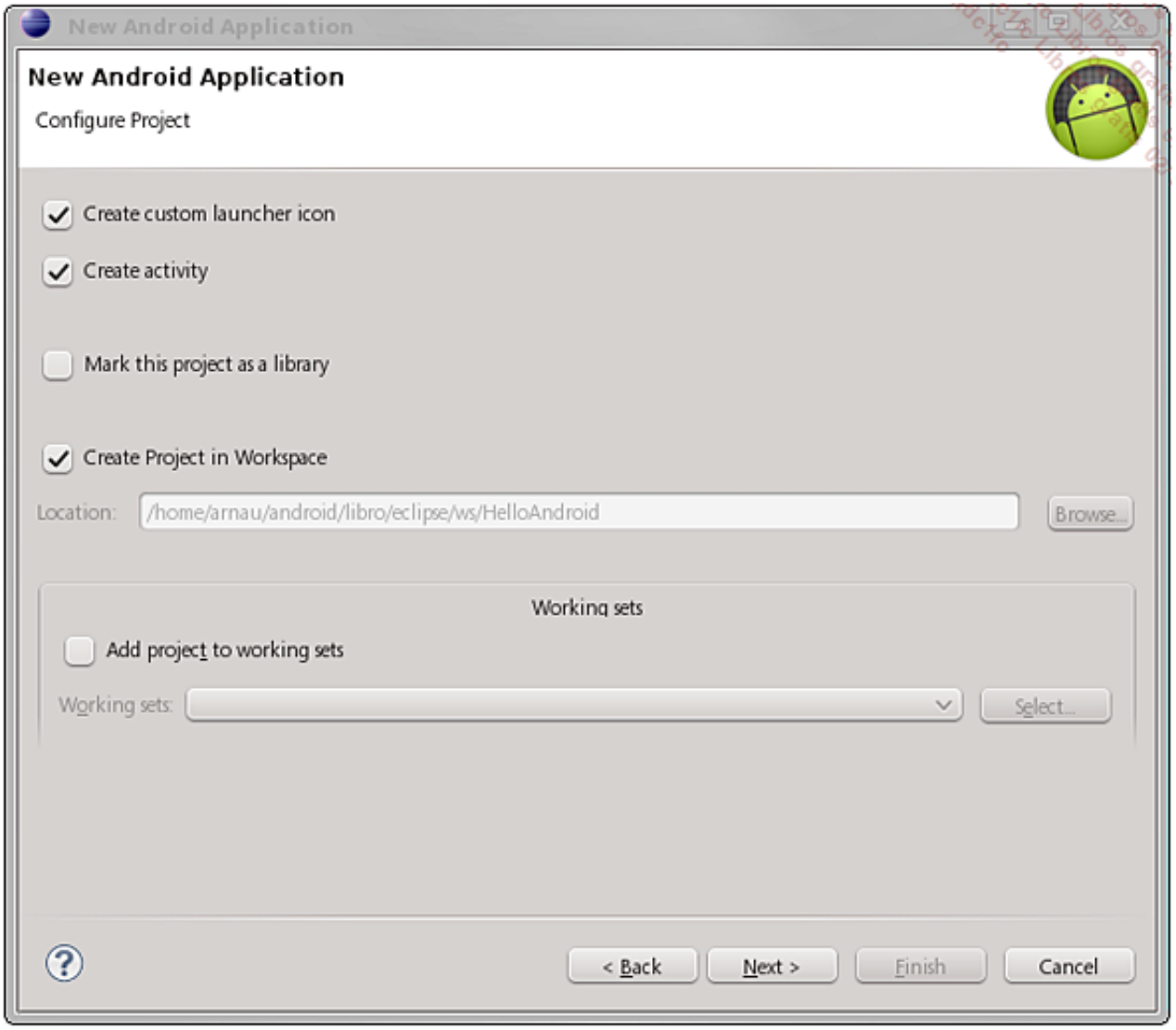
La primera pantalla le permite especificar:

- El nombre de la aplicación.
- El nombre del proyecto.
- El identificador de la aplicación (nombre del package).
- La versión mínima de Android compatible con la aplicación (**Minimum Required SDK**).
- La versión máxima de Android con la que se ha probado la aplicación (**Target SDK**).
- La versión de Android utilizada para desarrollar la aplicación (**Compile With**).
- El tema básico de la aplicación (**Theme**).

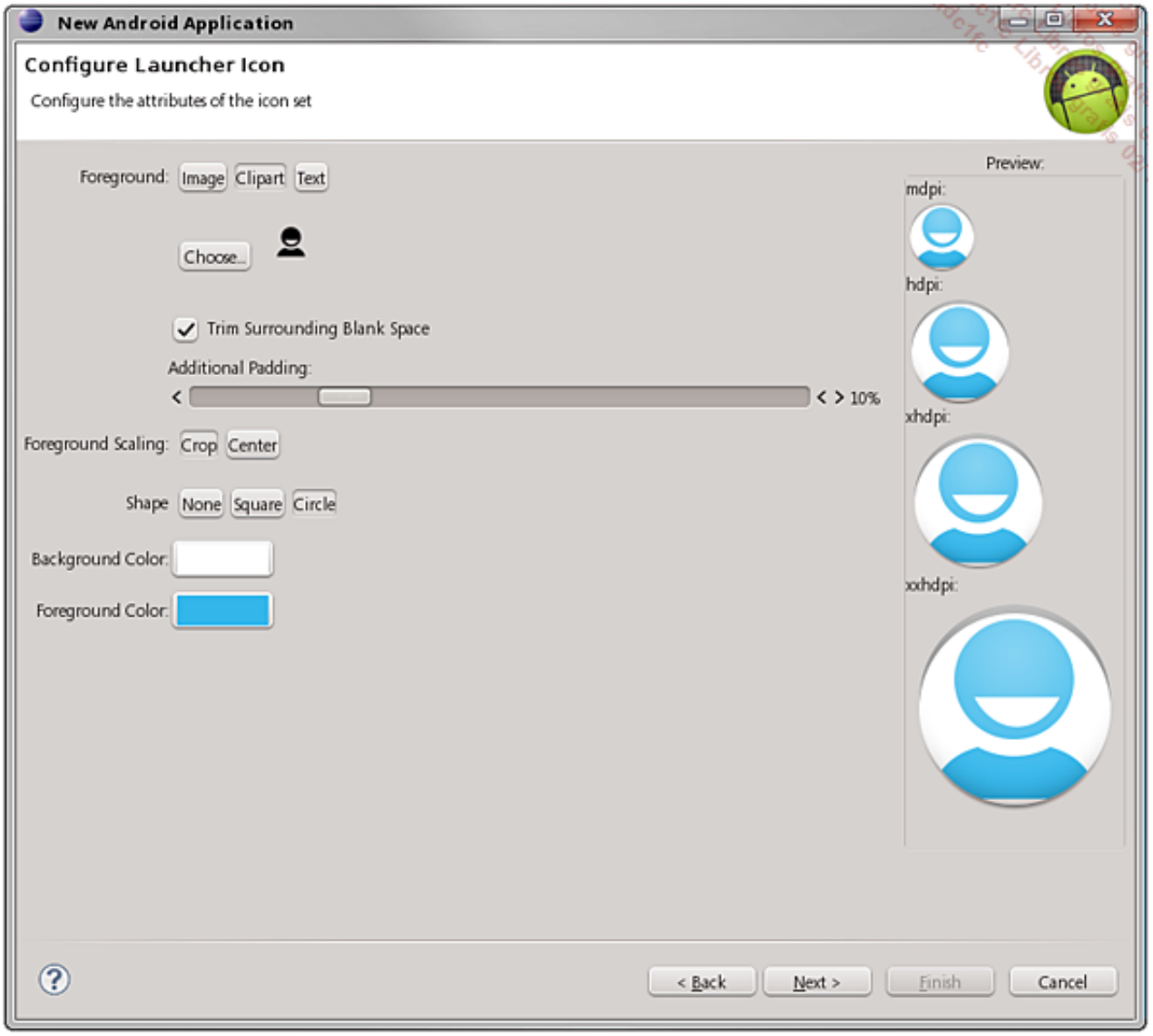


- Una vez se han introducido estos datos, haga clic en el botón **Next**. En esta pantalla podrá:
  - Definir, si lo desea, un icono personalizado (**Create custom launcher icon**).

- Crear la actividad (**Create activity**).
- Definir el comportamiento del proyecto (Aplicación Android o Librería Android) mediante la opción **Mark this project as a library**.
- Informar la ubicación del proyecto y el working set del proyecto.



Si ha seleccionado la opción que permite crear un icono personalizado, se abrirá la siguiente pantalla:



Dispone de varias opciones que le facilitan la creación del icono de su aplicación:

- Crear el icono desde una imagen personalizada.
- Crear el icono desde una imagen propuesta por el SDK Android.
- Crear el icono desde texto.

También puede especificar algunas propiedades del icono:

- Su padding (margen interno),
- Su posición,
- Su forma,
- Su color de fondo,
- Su color.

Puede saltarse este paso deseleccionando la opción **Create custom launcher icon** (en el paso anterior).

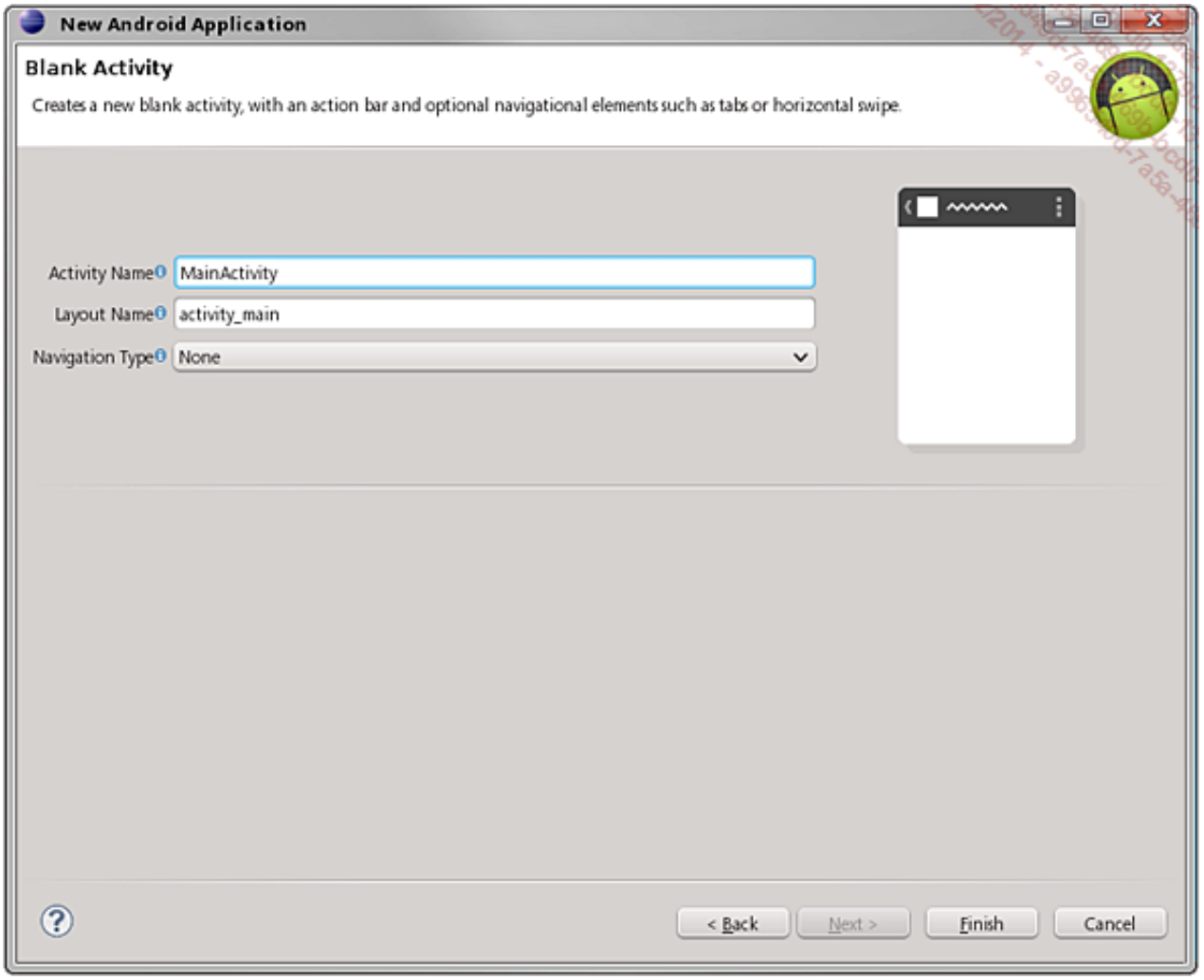
Una vez se ha creado el icono, haga clic en el botón **Next** para pasar a la pantalla que le permite especificar si desea crear una actividad inicial para su aplicación.

Si desea una actividad inicial, puede elegir entre una de estas opciones:

- Actividad vacía.

- Actividad a pantalla completa.
- Actividad maestro/detalle.

Elija la opción **Blank activity** (actividad vacía) y haga clic en el botón **Next**. Obtendrá una pantalla que le permitirá configurar las características específicas de su actividad.



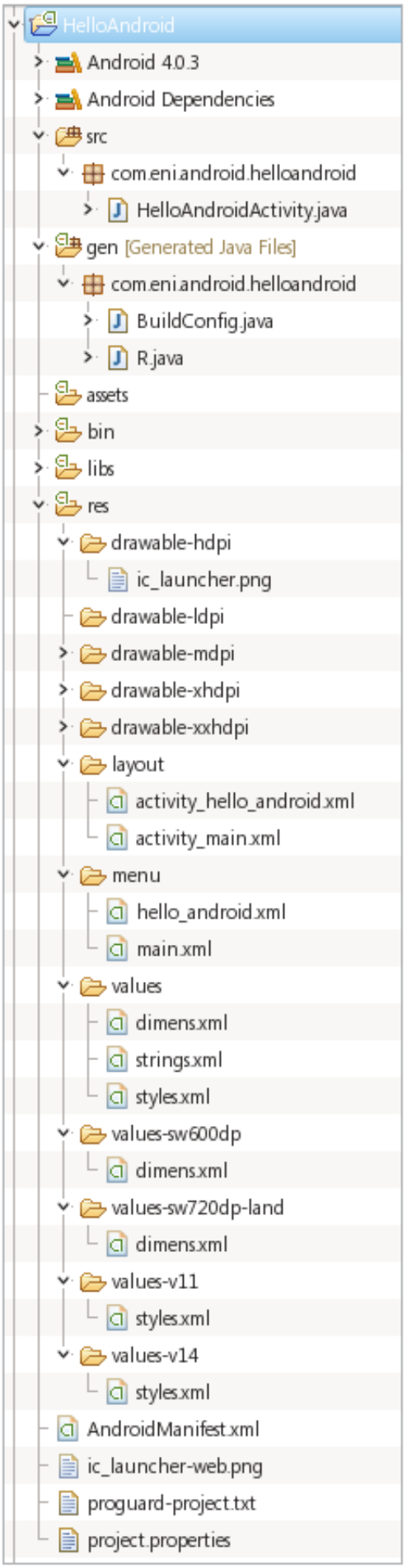
Esta nueva pantalla le permite indicar el estado inicial de su actividad:

- Nombre de la actividad (nombre del archivo).
- Título de la actividad.
- Nombre del archivo que sirve para crear la vista.
- Nombre de la clase madre de la actividad.
- Tipo de navegación que se usará en la actividad (Fixed tabs + Swipe, Scrollable tabs + Swipe o DropDown).

Para terminar la creación de su proyecto, haga clic en el botón **Finish**. El proyecto aparecerá en la pestaña **Package explorer**.

Arquitectura del proyecto

Desplegando el proyecto, descubrirá la arquitectura de una aplicación Android.



Una aplicación Android se compone de varias carpetas:

- **src**: esta carpeta contiene los fuentes de su aplicación (Actividades, Servicios, Código de capa de negocio, etc.).
- **gen**: esta carpeta es el resultado de la generación de archivos presentes en la carpeta **res** (de la voz inglesa Resources (Recursos)). El contenido de la carpeta Resources se precompila y se genera en la carpeta **gen** para que pueda utilizarlo fácilmente en su código fuente. Esta carpeta también contiene el archivo **BuildConfig** que le permite gestionar la visualización del log en modo depuración y desarrollo, todo ello gracias a una constante definida en este archivo.
- **assets**: esta carpeta almacena los datos o archivos que utiliza en su aplicación. Estos archivos no son ni dependientes de la resolución, ni del idioma del teléfono y se conserva su formato (no hay precompilación). Los archivos almacenados en la carpeta **assets** son de sólo lectura.
- **bin**: esta carpeta sirve para almacenar los binarios y los archivos de clase generados.
- **libs**: contiene las distintas librerías usadas por la aplicación.
- **res**: esta carpeta sirve para almacenar todos los recursos que utilice en su aplicación. Puede tener recursos diferentes en función de la resolución, de la orientación o del idioma del dispositivo. Los recursos se precompilan y se añaden al archivo **R.java** (véase el capítulo Creación de interfaces sencillas - Recursos).
  - **drawable-xxx**: contiene todas las imágenes en las distintas resoluciones aceptadas.
  - **layout-xxx**: contiene todas las vistas (modos horizontal y vertical).
  - **values-xxx**: contiene todos los archivos que contienen datos o valores (strings, arrays, colors, dimensions, styles, etc.).
  - **menus**: contiene las barras de acciones y menús de su aplicación.
  - **raw-xxx** : contiene archivos no compilables (archivo de audio, de video, etc.).
- **AndroidManifest.xml**: es el archivo de descripción de su aplicación.
- **proguard.cfg**: permite definir el sistema de ofuscación y de optimización de su código utilizado por ProGuard.
- **project.properties**: contiene información sobre el proyecto (versión y librerías referenciadas).

Explicaciones

A continuación, se realizará un recorrido por todos los archivos del proyecto para explicar su contenido.

1. Android Manifest

Para comenzar, abra el archivo Android Manifest (véase el capítulo Principios de programación - Manifiesto). Este archivo es bastante básico en este caso.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.eni.android.helloandroid"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
      android:minSdkVersion="8"
      android:targetSdkVersion="15" />

    <application
      android:allowBackup=”true”
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
      <activity
         android:name="com.eni.android.helloandroid.HelloAndroidActivity"
        android:label="@string/app_name" >
          <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
      </activity>
    </application>

</manifest>
```

En este archivo:

- El package (identificador) de su aplicación (en este caso com.eni.android. helloandroid) se define en el atributo **package** de la etiqueta manifiesto.
- La versión de la aplicación se detalla con los atributos **versionCode** (número de versión) y**versionName** (nombre de versión: visible en el Market) de la etiqueta manifiesto.
- La versión mínima del SDK para poder utilizar la aplicación se informa en el atributo**minSdkVersion** de la etiqueta **uses-sdk**, así como la versión del SDK utilizada para desarrollar la aplicación (atributo **targetSDKVersion**).
- El icono y el nombre de su aplicación se introducen con los atributos **icon** y **label** de la etiqueta **application** así, como el tema utilizado por la aplicación (véase el capítulo Personalización y gestión de eventos - Personalización).
- La descripción de la actividad principal (etiqueta **activity**) incluye:
  - El nombre de la clase que implementa Activity.
  - El título de la actividad.
  - Los filtros de la actividad:
    - **MAIN**: indica que se trata de la actividad principal de la aplicación.
    - **LAUNCHER**: indica que esta actividad está presente en el arranque de la aplicación.

2. Resources (recursos)

Drawable

La carpeta **drawable** contiene el icono de la aplicación en cuatro resoluciones (ldpi = baja resolución, mdpi = resolución media, hdpi = alta resolución, xhdpi = resolución muy alta).

Layout

La aplicación sólo contiene un valor, que sirve para mostrar la etiqueta "HelloAndroid".

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".HelloAndroidActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

</RelativeLayout>
```

Esta vista sólo contiene un RelativeLayout, que sirve de contenedor para el campo de texto (véase el capítulo Creación de interfaces sencillas - Layouts) cuyo contenido se declara en el archivo strings.xml (véase el capítulo Creación de interfaces sencillas - Recursos). El RelativeLayout tiene los atributos siguientes:

- Un margen interno (padding) para el texto.

  Este campo de texto también tiene los atributos siguientes:

- La anchura y la altura del elemento (véase el capítulo Creación de interfaces sencillas - Principios).

Values

Puede observar las cinco carpetas **values**:

- **values**: carpeta utilizada para almacenar diferentes valores útiles en una aplicación (utilizada en la mayoría de los casos).
- **values-sw600dp**: todos los archivos contenidos en esta carpeta reemplazan los archivos values por defecto cuando el dispositivo tiene una pantalla grande (tablet de 7 pulgadas, por ejemplo).
- **values-sw720dp-land**: todos los archivos contenidos en esta carpeta reemplazan los archivos values por defecto cuando el dispositivo tiene una pantalla muy grande (tablet de 10 pulgadas en horizontal, por ejemplo).
- **values-v11**: todos los archivos contenidos en esta carpeta reemplazan los archivos values por defecto cuando el dispositivo tiene la versión 3.x de Android (Honeycomb).
- **values-v14**: todos los archivos contenidos en esta carpeta reemplazan los archivos values por defecto cuando el dispositivo tiene una versión 4.x de Android (Ice Cream Sandwich o JellyBean).

La carpeta **values** contiene los archivos siguientes (véase el capítulo Creación de interfaces sencillas - Recursos):

- **strings.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">HelloAndroid</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>

</resources>
```

Este archivo contiene cuatro cadenas de caracteres:

  - El nombre de la aplicación.
  - El mensaje "Hello world!", que se mostrará.
  - La cadena de caracteres utilizada en la barra de acción/menú.
  - El título de la actividad principal.

- **dimens.xml**

```
<resources>

    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>

</resources>
```

Contiene las distintas dimensiones utilizadas en la aplicación (los distintos márgenes internos utilizados).

Se sobrescriben con el archivo **dimens.xml** que está ubicado en las carpetas **values-sw600dp** y **values-sw720dp-land** para especificar nuevas dimensiones para pantallas más grandes.

Contenido del archivo **dimens.xml** presente en la carpeta **values-sw600dp**:

```
<resources>

    <!--
         Customize dimensions originally defined in res/values/
dimens.xml (such as
         screen margins) for sw600dp devices (e.g. 7" tablets) here.
    -->

</resources>
```

Contenido del archivo **dimens.xml** presente en la carpeta **values-sw720dp-land**:

```
<resources>

    <!--
         Customize dimensions originally defined in res/values/
dimens.xml (such as
         screen margins) for sw720dp devices (e.g. 10" tablets)
in landscape here.
    -->
    <dimen name="activity_horizontal_margin">128dp</dimen>

</resources>
```

- **styles.xml**

```
<resources>

    <!--
```

```
        Base application theme, dependent on API level. This theme is
replaced
        by AppBaseTheme from res/values-vXX/styles.xml on newer devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Light">
        <!--
            Theme customizations available in newer API levels can go in
            res/values-vXX/styles.xml, while customizations related to
            backward-compatibility can go here.
        -->
    </style>

    <!-- Application theme. -->
    <style name="AppTheme" parent="AppBaseTheme">
        <!-- All customizations that are NOT specific to a particular
API-level can go here. -->
    </style>

</resources>
```

Contiene el tema utilizado por la aplicación (véase el capítulo Personalización y gestión de eventos - Personalización).

Este tema será distinto en aquellos dispositivos que dispongan de las API 11 (Honeycomb) y 14 (Ice Cream Sandwich o posterior). Todo ello para utilizar el tema Holo de Android, disponible solamente para las versiones 3.x y 4.x.

Contenido del archivo **styles.xml** presente en la carpeta **values-v11**:

```
<resources>

    <!--
        Base application theme for API 11+. This theme completely replaces
        AppBaseTheme from res/values/styles.xml on API 11+ devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Holo.Light">
        <!-- API 11 theme customizations can go here. -->
    </style>

</resources>
```

Contenido del archivo **styles.xml** presente en la carpeta **values-v14**:

```
<resources>

    <!--
        Base application theme for API 14+. This theme completely replaces
        AppBaseTheme from BOTH res/values/styles.xml and
        res/values-v11/styles.xml on API 14+ devices.
    -->
    <style name="AppBaseTheme"
parent="android:Theme.Holo.Light.DarkActionBar">
        <!-- API 14 theme customizations can go here. -->
    </style>

</resources>
```

3. Archivo generado

Esta carpeta contiene el archivo **R.java**. Se genera automáticamente con la precompilación de los archivos ubicados en la carpeta de recursos.

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
```

```
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package com.eni.android.helloandroid;

public final class R {
    public static final class attr {
    }
    public static final class dimen {
        /**  Default screen margins, per the Android Design guidelines.

         Customize dimensions originally defined in res/values/dimens.xml (such as
         screen margins) for sw720dp devices (e.g. 10" tablets) in landscape here.

         */
        public static final int activity_horizontal_margin=0x7f040000;
        public static final int activity_vertical_margin=0x7f040001;
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int action_settings=0x7f080000;
    }
    public static final class layout {
        public static final int activity_hello_android=0x7f030000;
        public static final int activity_main=0x7f030001;
    }
    public static final class menu {
        public static final int hello_android=0x7f070000;
        public static final int main=0x7f070001;
    }
    public static final class string {
        public static final int action_settings=0x7f050001;
        public static final int app_name=0x7f050000;
        public static final int hello_world=0x7f050002;
    }
    public static final class style {
        /**
        Base application theme, dependent on API level. This theme is
replaced
        by AppBaseTheme from res/values-vXX/styles.xml on newer devices.


            Theme customizations available in newer API levels can go in
            res/values-vXX/styles.xml, while customizations related to
            backward-compatibility can go here.


        Base application theme for API 11+. This theme completely replaces
        AppBaseTheme from res/values/styles.xml on API 11+ devices.

 API 11 theme customizations can go here.

        Base application theme for API 14+. This theme completely replaces
        AppBaseTheme from BOTH res/values/styles.xml and
        res/values-v11/styles.xml on API 14+ devices.

 API 14 theme customizations can go here.
         */
        public static final int AppBaseTheme=0x7f060000;
        /**  Application theme.
 All customizations that are NOT specific to a particular API-level can go
here.
```

```
         */
        public static final int AppTheme=0x7f060001;
    }
}
```


No modifique este archivo ya que sus modificaciones se perderán en la próxima compilación.

Puede observar que el archivo contiene referencias a todos los elementos presentes en la carpeta**res** (drawable, layout y strings).

Desde la nueva versión de ADT, la carpeta **gen** también contiene el archivo **BuildConfig.java**.

```
/** Automatically generated file. DO NOT MODIFY */ package com.eni.android.helloandroid;
 public final class BuildConfig {
    public final static boolean DEBUG = true;
}
```

Este archivo le permite gestionar si se encuentra en modo desarrollo o producción en su aplicación mediante la variable DEBUG.

El estado de esta variable se modifica automáticamente cuando genera su apk para que los logs no se muestren durante la ejecución de la aplicación en el dispositivo (véase el capítulo Depuración y gestión de errores - Logs).

4. Archivo fuente

Disponemos de un archivo "HelloAndroidActivity" que contiene la implementación de nuestra vista (véase el capítulo Creación de interfaces sencillas - Principios).

```
package com.eni.android.helloandroid;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
public class HelloAndroidActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_android);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.hello_android, menu);
        return true;
    }


}
```

Nuestra clase debe heredar de Activity y sobrecargar el método **onCreate** en el que se define el contenido de la vista gracias al método **setContentView**. La vista se obtiene gracias a la clase R.java, ya que se declara estáticamente en esta clase.

También contiene la creación de un menú utilizado en la aplicación (véase el capítulo Creación de interfaces sencillas - Menús).

Resultado

A continuación, ejecute el proyecto en el dispositivo objetivo (menú **Run** en Eclipse). El uso del plugin ADT permite instalar y ejecutar una aplicación en el dispositivo simplificando los siguientes pasos:

- Compilación y conversión del proyecto Android en un archivo ejecutable Android (.dex).
- Enpaquetar los archivos generados así como los recursos en un archivo apk.
- Arrancar el emulador.
- Instalación de la aplicación en el dispositivo objetivo.
- Inicio de la aplicación.

Con ello, obtendremos el siguiente resultado:

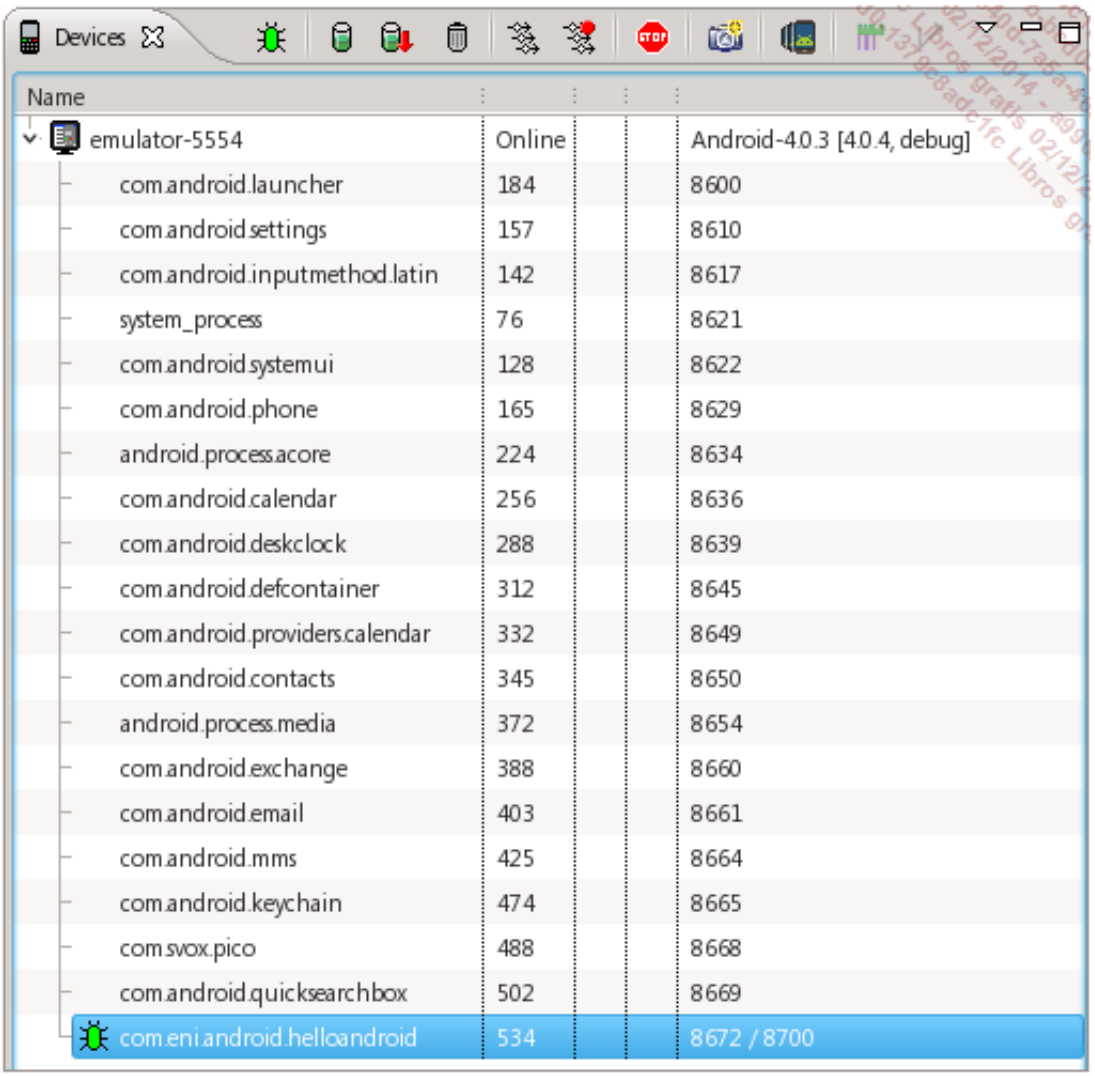
Principios

En el desarrollo de aplicaciones Android se encontrará con la necesidad de depurar código, corregir errores y realizar pruebas. El SDK Android incluye varias herramientas que facilitan estos pasos indispensables en la creación de aplicaciones Android.

DDMS (Dalvik Debug Monitor Server)

El DDMS es la herramienta de depuración incluida en el SDK Android, se compone de varios módulos:

- **Devices** (Dispositivos): este módulo permite:
  - Ver la lista de dispositivos conectados.
  - Ver la lista de procesos en ejecución en un dispositivo seleccionado.
  - Realizar capturas de pantalla de un dispositivo.
  - Activar la depuración en un proceso determinado.
  - Parar un proceso...



- **Emulator Control** (Control del emulador) permite:
  - Simular llamadas.
  - Simular mensajes.
  - Simular una localización...

Hay más pestañas que permiten:

- Obtener la lista de procesos en ejecución,
- Ver el logcat,
- Supervisar las asignaciones de memoria,
- Obtener estadísticas sobre el uso de la red,

- Explorar archivos…

Puede acceder al DDMS de dos maneras distintas:

- Desde Eclipse: haga clic en **Window - Open Perspective - Other** y, a continuación, seleccione**DDMS**.
- Desde la carpeta **tools** del SDK Android.

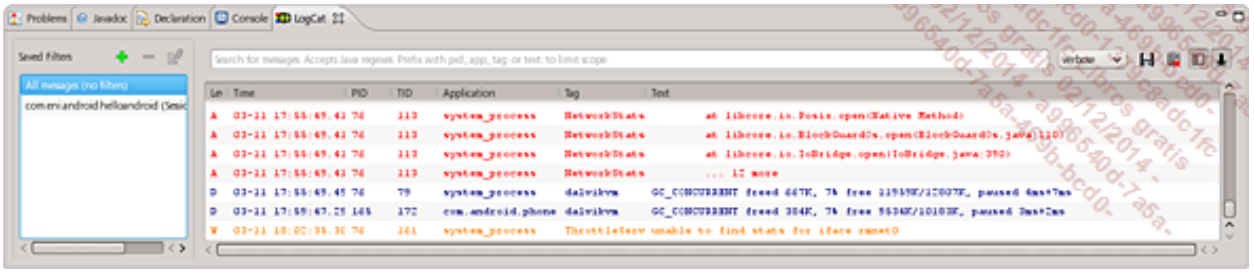
Logs

El sistema de logs es imprescindible en el desarrollo para Android. Es útil para mostrar mensajes que tienen diferentes niveles de importancia. Estos mensajes se pueden visualizar gracias a **Logcat**.

1. Logcat

Logcat es la interfaz que permite visualizar todos los mensajes que generan las distintas aplicaciones contenidas en el dispositivo.

Si la pestaña **LogCat** no aparece en Eclipse, puede mostrarla haciendo clic en **Window - Show view - Logcat**.



Con esta herramienta podrá:

- Ver los mensajes de log generados por el dispositivo seleccionado.
- Filtrar los mensajes que se muestran por aplicación, identificador o etiqueta de una aplicación.
- Buscar un mensaje.
- Filtrar los mensajes por nivel (verbose, debug, error, etc.).
- Guardar los logs seleccionados en un archivo.

Puede acceder a los logs por línea de comandos mediante la herramienta adb:

```
./adb logcat
```

2. Utilizar los logs

Puede mostrar un mensaje de log en cualquier parte de la aplicación gracias a la clase **Log**. Hay un método disponible para cada nivel de importancia del mensaje.

- **d (Debug - Depuración)**: sirve para mostrar un mensaje de depuración.
- **e (Error)**: sirve para mostrar un mensaje de error.
- **i (Info - Información):** sirve para mostrar un mensaje de información.
- **v (Verbose)**: sirve para mostrar un mensaje verbose.
- **w (Warning)**: sirve para mostrar una advertencia.
- **wtf (What a Terrible Failure)**: sirve para mostrar un error que nunca debería ocurrir.

A continuación se muestra el prototipo para un mensaje de error:

```
Log.e(String tag, String message);
```

o

```
Log.e(String tag, String message, Throwable tr);
```


Puede incluir una excepción en los logs mediante el tercer parámetro del método de Log.

A continuación se muestra un ejemplo de visualización de un mensaje de error:

```
private static final String TAG = "MiActivity";
...
Log.e(TAG, "Mi mensaje de error");
```


En general, el log mostrado tendrá como TAG el nombre de la actividad en la que se encuentra. Esto nos permitirá localizar más fácilmente el origen del mensaje.

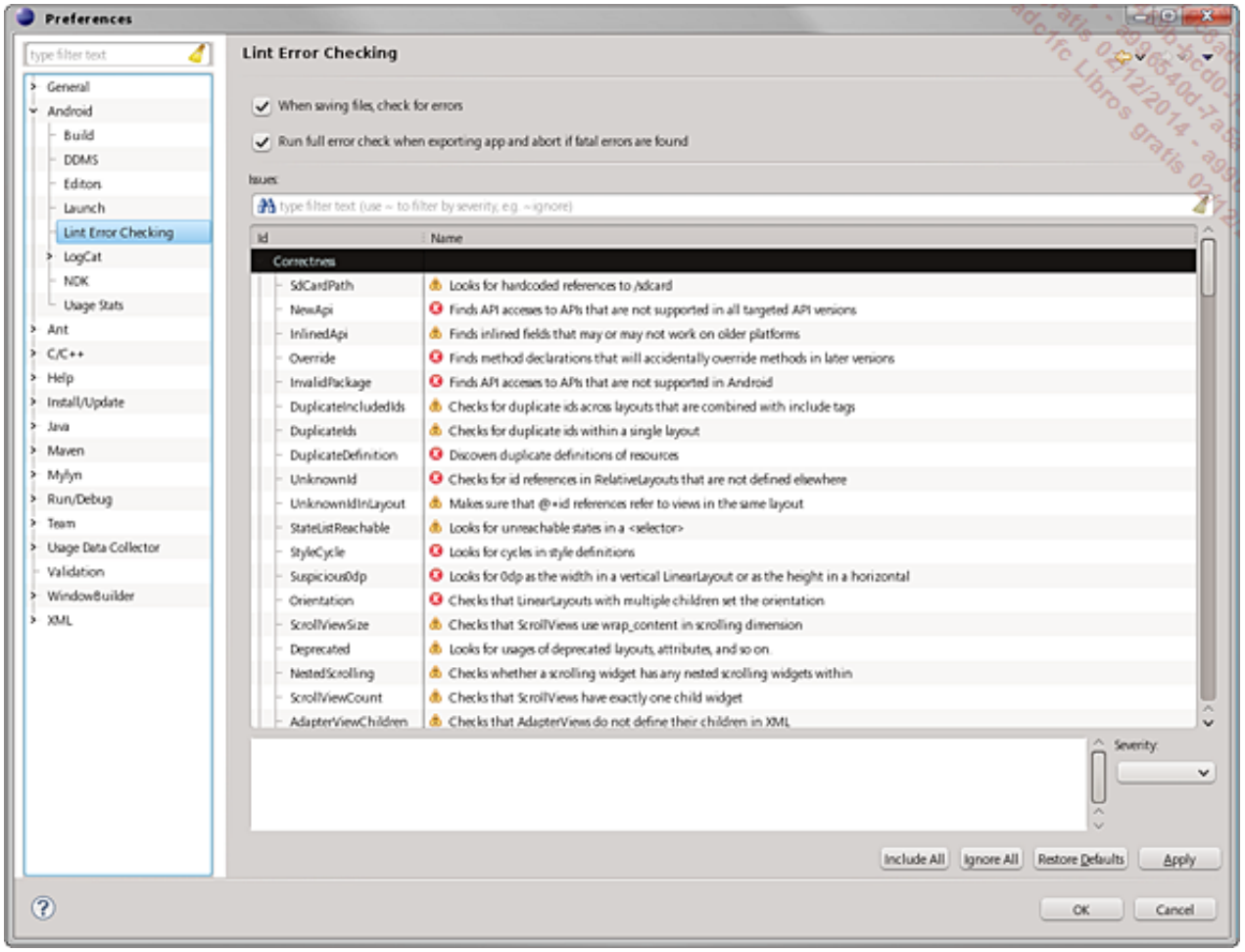
Tiene la posibilidad de utilizar el flag **debug** disponible en el archivo **BuildConfig** para mostrar los logs únicamente durante la fase de desarrollo y de test. En la generación del apk firmado, se eliminarán los logs del código generado. Esta buena práctica permite limitar las interacciones entre una aplicación y el dispositivo.

```
if (BuildConfig.DEBUG)
   Log.v("HelloAndroidActivity", "Log de mi aplicación");
```

Android Lint

Esta herramienta le permite especificar una lista de reglas que se aplicarán a sus proyectos Android (recursos no internacionalizados, utilización de APIs no soportada en todas las versiones de Android, uso de dimensiones en px, etc.).

Puede configurar la lista de reglas que se tendrán en consideración en la pantalla de configuración de Eclipse (apartado **Lint Error Checking** en la sección **Android** de la configuración de Eclipse).



Esta herramienta es muy importante en la fase de desarrollo porque permite comprobar que sus implementaciones siguen las buenas prácticas de desarrollo Android.

Los errores Lint se muestran en la ventana **Lint Warnings**, accesible haciendo clic en **Window - Show View - Other - Android - Lint Warnings**.

ADB (Android Debug Bridge)

ADB es una herramienta muy útil e importante en el desarrollo para Android. En esta sección, descubrirá algunas funcionalidades prácticas de adb:

Listar dispositivos

Puede listar los dispositivos Android detectados por el SDK Android por línea de comandos mediante el siguiente comando:

```
./adb devices
016B7DFE0101001A   device
emulator-5554      online
```

Si encuentra problemas con el uso de la herramienta adb, puede reiniciarla con los comandos adb kill-server y adb start-server.

Conectarse a un dispositivo

Conectarse a un dispositivo detectado es muy sencillo, puede utilizar el comando siguiente:

```
./adb shell
```

Si tiene varios dispositivos conectados, debe especificar el dispositivo deseado mediante el argumento `-s`.

```
adb -s emulator-5554 shell
```

Una vez el comando se ha ejecutado con éxito, está conectado al shell del dispositivo seleccionado y tiene la posibilidad de ejecutar los comandos que desee (ls, cd, etc.).

Instalar una aplicación

Puede instalar directamente una aplicación en el dispositivo mediante el comando.

```
adb -s emulator-5556 install miAplicación.apk
```

También puede desinstalar una aplicación.

```
adb -s emulator-5556 uninstall com.eni.android.application
```

La desinstalación se realiza con el nombre del paquete de la aplicación y no con el nombre del archivo apk.

La obtención del nombre del package (así como otros datos) de una aplicación a partir de un apk se realiza con la herramienta **aapt** (presente en la carpeta **platform-tools** del SDK Android):

```
aapt d badgind miarchivo.apk
```

Copiar archivos

Puede copiar archivos/carpetas desde un dispositivo a un PC mediante el comando **pull**.

```
adb pull /sdcard/miArchivo miArchivo
```

También puede copiar un archivo desde el PC a un dispositivo mediante el comando **push**.

```
adb push miArchivo /sdcard/miArchivo
```

Estos comandos son sólo un pequeño aperitivo de las posibilidades que ofrece adb, puede conocer el resto de funcionalidades mediante el comando:

```
./adb help
```

Depuración paso a paso

Para depurar su aplicación y comprender el origen de los errores, podrá utilizar puntos de ruptura (Breakpoints). Estos breakpoints le permitirán ver la ejecución de su aplicación paso a paso, comprender el origen de los errores y ver la pila de llamadas y el estado de sus variables durante la ejecución.

- Debe ejecutar el proyecto en modo depuración (haga clic con el botón derecho sobre el proyecto deseado y seleccione **Debug As - Android Application**).

```
public class HelloAndroidActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_android);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.hello_android, menu);
        return true;
    }

}
```

- No es posible ubicar breakpoints en los recursos de una aplicación.

Interacción con el emulador

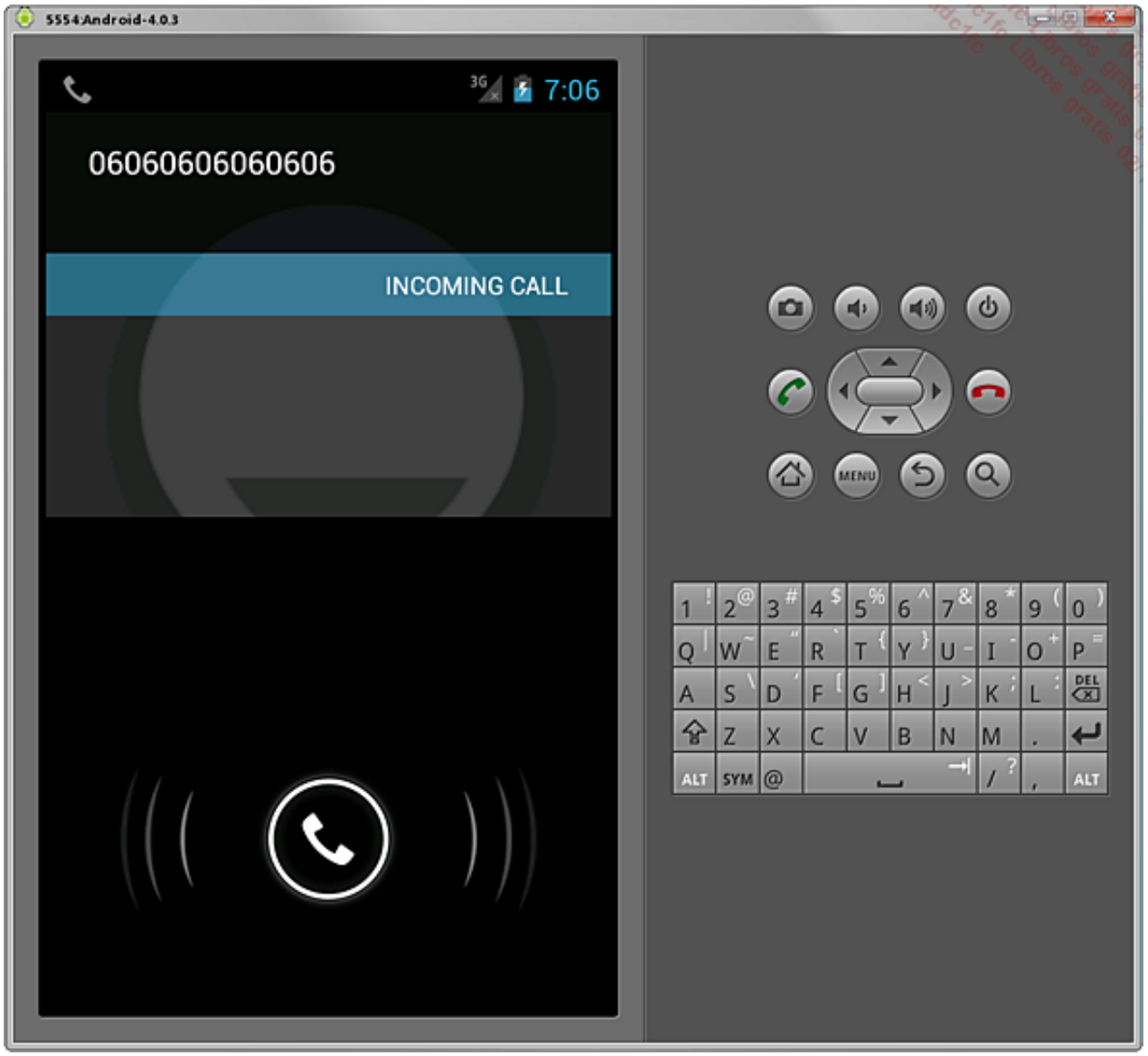
Puede interactuar con el emulador para simular el uso del dispositivo. Esta funcionalidad es posible gracias al **Emulator Control** (Control del emulador).

- Para mostrar el **Emulator Control**, haga clic en **Window - Show View - Other** y, a continuación, seleccione **Emulator Control** o vaya simplemente a la perspectiva DDMS.

1. Simular llamadas

Para simular una llamada, utilice la sección **Telephony Actions** del **Emulator Control**.

- Introduzca un número de teléfono en el campo **Incoming number** (Número origen) y, a continuación, pulse el botón **Call** (Llamar).



2. Simular mensajes

Para simular la recepción de un mensaje hay que utilizar la sección **Telephony Actions** como se ha explicado anteriormente.

- Una vez se ha introducido el número, active la selección de **SMS** e introduzca el contenido del mensaje y, a continuación, pulse el botón **Send** (Enviar).

3. Simular una posición GPS

Para simular una posición GPS y, por lo tanto, una localización del usuario, utilice la sección**Locations Controls** (Controles de localización).

- Introduzca una latitud y una longitud y, a continuación, pulse el botón **Send** para enviar la posición al teléfono (véase el capítulo Google Maps y geolocalización - Localización).

4. Realizar capturas de pantalla

La vista **Devices** (Dispositivos) le permite, entre otras acciones, realizar capturas de pantalla.

Para mostrar esta vista, haga clic en **Window - Show view - Other**, a continuación seleccione**Devices** (Dispositivos) o simplemente acceda a la perspectiva DDMS.

- Seleccione un dispositivo y haga clic en el pequeño icono que representa una cámara de fotos.

Pruebas en el teléfono

Si tiene un teléfono Android, puede probar sus aplicaciones en él, además de las pruebas realizadas con los distintos emuladores.

1. Utilizar su teléfono

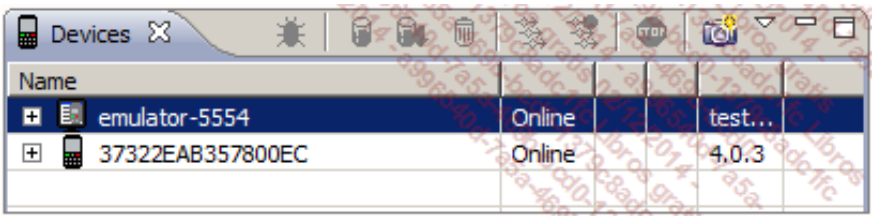
Para utilizar su teléfono, debe realizar algunas operaciones más o menos largas, en función del dispositivo y de su OS.

Si está en Linux/Mac, no necesita realizar ninguna operación para hacer que Android reconozca su teléfono.

En Windows:

- Si dispone de un teléfono Google (Nexus One, Nexus S y Galaxy Nexus), los drivers de uso de sus teléfonos están incluidos en el SDK Android (véase el capítulo El entorno de desarrollo - Instalación del entorno Java).
- Si dispone de un teléfono de otra marca (HTC, Samsung, etc.), deberá descargara los drivers disponibles del sitio del fabricante.

Para saber si puede utilizar su teléfono para probar sus aplicaciones (configuración realizada con éxito), inicie la vista DDMS de Eclipse y compruebe, en la sección **Devices** (Dispositivos), si su teléfono aparece en la lista de dispositivos.



2. Opciones de desarrollo

Todo dispositivo Android (versión 4.0 como mínimo) tiene opciones para desarrolladores. Estas opciones están disponibles en la categoría **Opciones de desarrollador**, en los ajustes de su teléfono.

No olvide activar la **Depuración USB** para utilizar su dispositivo a través de Eclipse.

A continuación se muestran las opciones más útiles:

Depuración USB: activa el modo de depuración cuando el teléfono está conectado al PC.

Permitir ubicaciones falsas: permite la simulación de posiciones ficticias y, por lo tanto, simular una posición distinta a la del dipositivo.

Modo estricto: esta opción es muy util, ya que habilita que la pantalla de su dispositivo destelle en rojo cuando una operación larga no se realice en un thread distinto al del UI (véase el capítulo Tratamiento en tareas en segundo plano).

Mostrar uso de CPU: muestra un mensaje por pantalla que indica la cantidad de CPU utilizada.

No mantener actividades: permite eliminar inmediatamente las actividades abandonadas y sólo conservar una actividad a la vez. Esta funcionalidad le permite probar al límite su aplicación y su comportamiento cuando se finaliza una actividad anterior. Esta funcionalidad permite, entre otros,

comprobar cómo se guarda el estado de sus actividades (**onSaveInstance** y **onRestoreInstance**).

Limitar procesos en segundo plano: permite definir el número máximo de procesos en segundo plano, es decir, el número de aplicaciones que pueden conservarse en segundo plano.

Pruebas unitarias

La creación de pruebas unitarias es un elemento esencial en cada proyecto Android. Para ello, Android proporciona un sistema de pruebas unitarias bastante completo que le permitirá probar sus aplicaciones.

Para empezar, cree un proyecto de pruebas que se aplique al proyecto **HelloAndroid** (creado en el capítulo Mi primera aplicación: HelloAndroid) para probar la exactitud de la cadena de caracteres mostrada.

- Para crear el proyecto de pruebas, haga clic en **File - New - Other** y, a continuación, seleccione**Android Test Project**.

La primera pantalla permite introducir el nombre del proyecto (por convenio, el nombre del proyecto de pruebas se corresponde con el nombre del proyecto concatenado con la palabra Test).

- A continuación, haga clic en **Next** para seleccionar el proyecto objeto de las pruebas (en este caso **Cap4_HelloAndroid**). Haga clic, a continuación, en **Next** y seleccione la versión 4.0.3 de Android. Para finalizar, haga clic en **Finish**.

El siguiente paso consiste en crear una clase de pruebas. Esta clase puede sobrecargar varios tipos de clases en función del componente que desee probar.

Con el objetivo de probar una actividad, hay que sobrecargar la clase**ActivityInstrumentationTestCase2**. Esta clase debe parametrizarse con la clase correspondiente a la actividad que se va a probar.

Con lo que tendremos:

```
public class HelloAndroidActivityTest extends
ActivityInstrumentationTestCase2<HelloAndroidActivity>
```

A continuación, deberá sobrecargar los siguientes dos métodos:

- Un constructor para la clase de pruebas.
- El método **setUp()**, que permite inicializar el entorno de pruebas (por ejemplo, inicializar el juego de pruebas que se usará en el test).

Y crear los métodos que implementan las distintas pruebas que deseamos realizar:

- El método que permite probar las precondiciones de la prueba (por ejemplo, una inicialización correcta de la vista).
- Los métodos que ejecutan sus pruebas.

El constructor es muy sencillo y permite inicializar la clase usando la clase madre. Debe especificar la clase que se quiere probar.

```
public HelloAndroidActivityTest() {
   super(HelloAndroidActivity.class);
}
```

El método **setUp** permite inicializar el entorno utilizado para las pruebas. En este ejemplo, hay que inicializar:

- La actividad (Activity).
- El campo de texto (TextView).
- La cadena de texto utilizada como referencia.

```
@Override
```

```
protected void setUp() throws Exception {
   super.setUp();
   myActivity = this.getActivity();
   myView = (TextView)
myActivity.findViewById(com.eni.android.helloandroid.R.id.helloTxt);
   expectedStr =
myActivity.getResources().getString(com.eni.android.helloandroid.R.
string.hello_world);
}
```

Puede observar que los recursos utilizados se corresponden con los recursos del proyecto que se está probando.

Debe crear un método que permita comprobar las precondiciones de las pruebas. Estas precondiciones, generalmente, corresponden a la cadena de caracteres de referencia.

Con lo que tenemos:

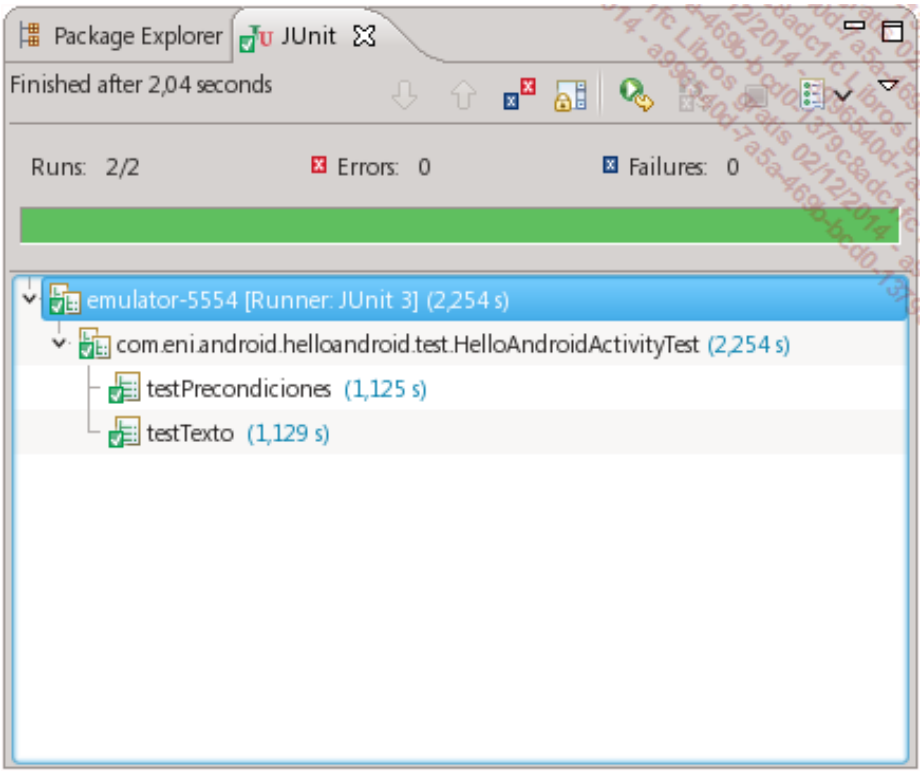
```
public void testPreconditions() {
   assertNotNull(myActivity);
   assertNotNull(myView);
   assertNotNull(expectedStr);
}
```

En este ejemplo se comprueba que las distintas variables útiles para el test no son iguales a null (inicialización realizada con éxito).

Para terminar, hay que crear el método que permite realizar la prueba. Esta prueba consiste en comprobar que el campo de texto es igual a la cadena de caracteres esperada.

```
public void testTexto() {
   assertEquals(expectedStr, (String) myView.getText());
}
```

- Una vez creada la clase, haga clic con el botón derecho del ratón sobre el proyecto de pruebas. A continuación, seleccione **Run As - Android JUnit Test**. Se abrirá una ventana automáticamente para mostrar el resultado de los tests. Obtendremos:


Package Explorer
JUnit
Finished after 2,04 seconds
Runs: 2/2
Errors: 0
Failures: 0
emulator-5554 [Runner: JUnit 3] (2,254 s)
com.eni.android.helloandroid.test.HelloAndroidActivityTest (2,254 s)
testPrecondiciones (1,125 s)
testTexto (1,129 s)

Pruebas de interfaces

Puede probar las interfaces de una aplicación, así como sus comportamientos, mediante varias herramientas que presentamos en esta sección.

1. Monkey

Monkey es una herramienta incluida en el SDK Android. Permite probar una aplicación mediante eventos aleatorios. El objetivo es realizar una prueba de estrés que simule acciones de usuario.

Esta herramienta, que se puede usar por línea de comandos a través de la herramienta **adb**, se encuentra en la carpeta **platform-tools** del SDK Android.

Monkey incluye un gran número de opciones:

- Número de eventos que se simularán.
- Tipo y frecuencia de los eventos que se probarán.
- Posibilidad de depurar.

Detendrá las tareas descritas si:

- La lista de tareas ha finalizado.
- La aplicación produce un error (la herramienta mostrará los errores encontrados).
- La aplicación deja de responder (la herramienta mostrará los errores encontrados).

A continuación se muestra la sintaxis básica de Monkey:

```
adb shell monkey [opciones] <número_de_eventos>
```

Debe especificar el identificador del paquete correspondiente a la aplicación que se desea probar.

A continuación se muestra un ejemplo de uso básico de Monkey, para producir 500 eventos aleatorios en una aplicación instalada en un dispositivo determinado.

```
adb shell monkey -p com.eni.android.actionbar -v 500
```

Puede especificar el tipo de elemento que desea ejecutar mediante las distintas opciones que se pueden consultar en la página de ayuda del comando.

```
adb shell monkey --help
```

2. Robotium

Robotium es un framework de test parecido a Selenium. Permite automatizar las pruebas de interfaz de Android.

Gracias a este framework, puede crear pruebas automatizadas y robustas para sus actividades. Para ello, simplemente hay que incluir el archivo jar de la librería disponible en el sitio web oficial de Robotium: http://code.google.com/p/robotium/

También permite la creación de pruebas de interfaz con un mínimo de conocimiento de la aplicación que se desea probar.

Principios

La creación de una interfaz en Android puede realizarse de dos formas:

- **Creación estática**: se realiza en XML.
- **Creación dinámica**: se realiza en Java.

Ambos métodos se pueden combinar para crear interfaces más ricas (véase el capítulo Creación de interfaces avanzadas - Interfaces dinámicas).

Una interfaz se compone de:

- **Uno o varios archivos XML**: representan la parte estática de una interfaz. Está formada por los distintos elementos (Botón, Texto, Campo de edición, etc.).
- **Un archivo JAVA (Actividad)**: representa la parte dinámica de una interfaz, las interacciones y tratamientos que hay que realizar, etc.

La clase que representa la base de todos los elementos de una interfaz en Android es la clase **View**.

Todos los elementos básicos de una vista (botón, campo de texto...) heredan de esta clase.

La modificación de una vista se puede realizar de dos formas:

- Actualizando el código XML de la interfaz (pestaña correspondiente al nombre del archivo).
- Actualizando la vista mediante el editor de interfaces (pestaña **GraphicalLayout**).

1. Declarar identificadores

Un identificador se corresponde con un nombre único asignado a un elemento de una interfaz. Gracias a este identificador, puede implementar las interacciones y los tratamientos para cada elemento de la interfaz.

Para asociar un identificador a un elemento de una interfaz, hay que utilizar el siguiente atributo:

```
android:id="@+id/nombre_identificador"
```

La declaración de un identificador se compone de varios elementos:

- **android:id**: nombre del atributo.
- **@+**: indica la declaración de un nuevo identificador.
- **id**: corresponde a la categoría del identificador (en general, puede usar **id**, pero no es obligatorio).
- **nombre_identificador**: corresponde al identificador de un elemento.

La sintaxis siguiente permite acceder al identificador de un elemento desde un archivo Java:

```
R.id.nombre_identificador
```

o desde un archivo XML:

```
@id/nombre_identificador
```

2. Combinar con actividades

Una vez la parte estática de una interfaz se ha declarado, hay que crear una clase Java que represente su actividad.

Cada vez que se cree una nueva interfaz, ésta deberá declararse en el manifiesto de la aplicación.

Esta clase debe:

- Heredar de la clase **Activity**.
- Sobrecargar al menos el método ***onCreate*** (véase el capítulo Principios de programación - Ciclo de vida de una actividad).
- Enlazar la actividad con la interfaz mediante el método ***setContentView***.

Para crear una nueva actividad, haga clic con el botón derecho del ratón en el proyecto y seleccione la opción **New - Other - Android Activity**.

Por ejemplo, tomemos una interfaz creada en el archivo ***home.xml***. Para poder asociarla a una actividad, el método ***onCreate*** debe contener como mínimo el siguiente código:

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.home);
}
```

Puede observar el uso del archivo R.java para obtener el layout deseado.

No olvide declarar su actividad en el archivo de manifiesto de su aplicación. La declaración de nuevos componentes (actividad, servicio...) se realiza entre las etiquetas **<activity>**.

```
<activity android:name="ruta.package.MyNewActivity"
android:label="@string/activity_title">
```

Puede especificarle propiedades o comportamientos mediante filtros de intención. Los filtros de intención se dividen en varias categorías:

- **Acciones (etiqueta action)**: especifican acciones (comportamientos) de un componente. Por ejemplo: **ACTION_CALL** (puede pasar una llamada telefónica), **ACTION_MAIN** (actividad principal de la aplicación), **ACTION_SYNC** (sincroniza datos entre el dispositivo y un servidor), etc. También puede crear sus propias acciones.
- **Datos (etiqueta data)**: especifican el tipo de datos tratado por el componente.
- **Categorías (etiqueta category)**: especifican la categoría del componente. Por ejemplo:**CATEGORY_BROWSABLE** (el navegador lo puede invocar para mostrar datos),**CATEGORY_LAUNCHER** (la actividad estará disponible desde el lanzador de la aplicación), etc.
- Extras: representa datos adicionales que serán facilitados a la actividad.
- Diferentes flags útiles para la actividad.

Por ejemplo:

```
<activity android:name=".MyActivity"
          android:label="@string/activity_title">
```

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>

</activity>
```

3. Especificar el tamaño de los elementos

En cada declaración de un elemento de una vista (layout o componente), debe especificar su altura y su anchura (***android:height*** y ***android:width***).

Puede especificar estos valores de varias maneras:

- ***match_parent*** (anteriormente fill_parent): significa que el tamaño del elemento es igual al del elemento padre.

  Por ejemplo, un botón que tenga definida la anchura a un valor match_parent ocupará el mismo espacio que su contenedor.

- ***wrap_content***: significa que el tamaño del elemento es igual al de su contenedor.

  Por ejemplo, un botón que tenga la anchura definida a wrap_content tendrá el tamaño de la suma del tamaño de su contenido más el de los diferentes espacios internos.

- ***Especificando un valor***: puede definir el tamaño de un elemento con constantes.

Hay que especificar el tamaño de los elementos en **dp (density-independent pixels)** y no en px. Los tamaños especificados en dp conservan las mismas proporciones sea cual sea la densidad de pantalla.

Layouts

Los layouts facilitan la organización de los distintos elementos que componen una interfaz. Sirven de contenedor a los componentes de una vista. Todos los layouts Android heredan de la clase**ViewGroup**.

La clase **ViewGroup** hereda de la clase **View**.

1. FrameLayout

El FrameLayout es el layout más sencillo, representa un espacio que se puede rellenar con cualquier objeto a su elección.

Todo elemento añadido a un FrameLayout se posiciona respecto a la esquina superior izquierda del layout. Puede cambiar esta posición mediante el atributo **gravity**.

Tiene la posibilidad de añadir varios elementos en un mismo FrameLayout y modificar la visibilidad de estos elementos para mostrarlos u ocultarlos.

Se utiliza generalmente el FrameLayout para realizar la superposición de elementos en una interfaz.

2. LinearLayout

El LinearLayout permite alinear elementos (en el orden de las declaraciones) en una dirección (vertical u horizontal).

Puede definir los atributos siguientes:

- Orientación del layout (específica a este layout).
- Gravedad de los elementos.
- Peso de los elementos.

a. Orientación

En la creación de un LinearLayout, debe detallar su orientación (horizontal o vertical) mediante el uso del atributo ***android:orientation***.

La orientación por defecto es la horizontal.

b. Posicionamiento de un elemento

Para definir el posicionamiento de un elemento en un LinearLayout, hay dos atributos disponibles:

- **layout_gravity**: especifica el posicionamiento de un elemento en su contenedor.
- **gravity**: especifica el posicionamiento del contenido de un elemento (por ejemplo, se puede especificar la posición de un texto en un botón).

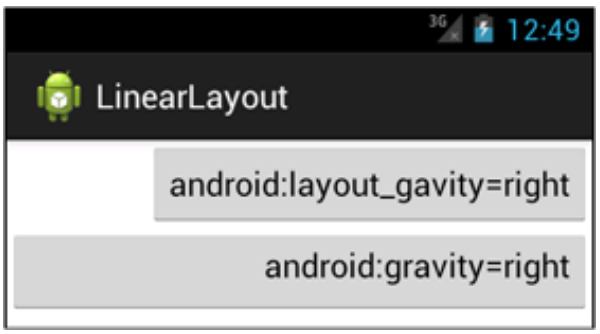
A continuación se muestra un ejemplo con ambos atributos:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="vertical" >

      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/layout_gravity"
          android:layout_gravity="right" />

      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/gravity"
          ìandroid:gravity="right" />

</LinearLayout>
```

Esta vista se compone de un:

- Layout, definido con una orientación vertical.
- Botón A, que tiene una anchura igual a la de su contenido. Este botón se posiciona a la derecha del layout (**layout_gravity**).
- Botón B, que tiene un texto posicionado a la derecha del botón (**gravity**).



c. Peso de un elemento

El peso sirve para indicar a un elemento el espacio que puede ocupar. Cuanto mayor sea el peso de un elemento, más se podrá extender un componente y ocupar el espacio disponible. El tamaño del espacio que desea alargar debe ser de **0px**.

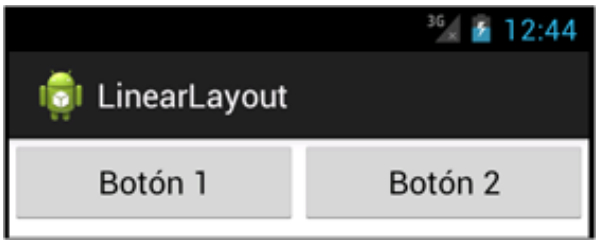
El valor por defecto del peso de un elemento es 0.

El peso de dos elementos debe tener el mismo valor para que cada uno de ellos tenga el 50% del espacio disponible.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="horizontal" >

      <Button
          android:layout_width="0px"
          android:layout_height="wrap_content"
```

```
            android:text="@string/btn1"
            android:layout_weight="1" />

        <Button
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:text="@string/btn2"
            android:layout_weight="1" />

</LinearLayout>
```

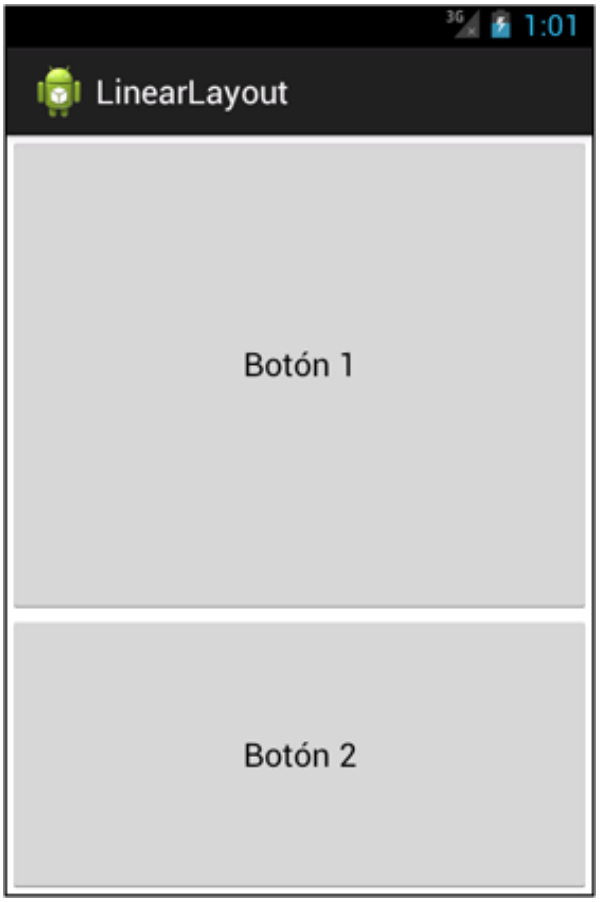
Con lo que obtenemos:



El ejemplo siguiente representa dos botones alineados de forma vertical. Sin embargo, el botón 1 es el doble de grande que el botón 2. Puede observar que el peso del botón 1 es también el doble que el del botón 2.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >

        <Button
            android:layout_width="fill_parent"
            android:layout_height="0px"
            android:text="@string/btn1"
            android:layout_weight="2" />

        <Button
            android:layout_width="fill_parent"
            android:layout_height="0px"
            android:text="@string/btn2"
            android:layout_weight="1" />

</LinearLayout>
```

Con lo que obtenemos:



3. TableLayout

El TableLayout posiciona todos sus hijos en filas y columnas, en forma de tabla. Ofrece la posibilidad de dejar celdas vacías, pero no la de extenderse en varias filas o en varias columnas.

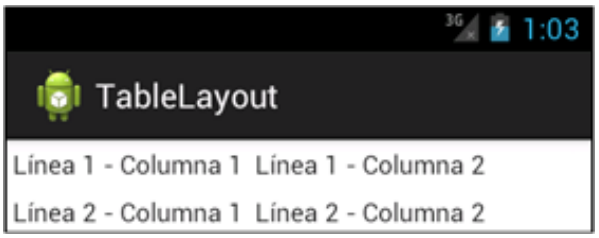
Para crear filas en un TableLayout, utilice el elemento **TableRow**. Cada elemento de tipo TableRow representa un elemento con 0 o más columnas en su interior.

A continuación se muestra un ejemplo de una vista que tiene dos filas, cada una de ellas compuesta por dos elementos (dos columnas):

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <TableRow>
        <TextView
            android:padding="3dp"
            android:text="@string/line1_column1" />
        <TextView
            android:padding="3dp"
            android:text="@string/line1_column2" />
    </TableRow>

    <TableRow>
        <TextView
            android:padding="3dp"
            android:text="@string/line2_column1" />
        <TextView
            android:padding="3dp"
            android:text="@string/line2_column2" />
```

```
    </TableRow>
</TableLayout>
```

Con lo que obtenemos:



El atributo **padding** permite especificar el margen interno de un elemento (el espacio entre los bordes del elemento y su contenido). En cambio, el atributo **margin**, permite especificar el margen externo de un elemento.

4. RelativeLayout

Este layout permite ubicar elementos unos en función de los otros. Un elemento puede posicionarse en función de su contenedor o en función de otro elemento.

a. Posicionamiento relativo al contenedor

Puede posicionar un elemento en función de los bordes del contenedor: para ello, existen cuatro atributos que permiten posicionar un elemento en función de sus cuatro bordes (arriba, abajo, derecha e izquierda).

Para ello, utilice uno o varios de los atributos siguientes con un valor booleano true/false (verdadero/falso).

- ***android:layout_alignParentTop***: alinear el elemento con el borde superior del contenedor.
- ***android:layout_alignParentBottom***: alinear el elemento con el borde inferior del contenedor.
- ***android:layout_alignParentLeft***: alinear el elemento con el borde izquierdo del contenedor.
- ***android:layout_alignParentRight***: alinear el elemento con el borde derecho del contenedor.

Puede combinar varios valores de posicionamiento.

b. Posicionamiento relativo a otros elementos

Dispone de nueve opciones de posicionamiento distintas. Para utilizarlas, añada el atributo deseado indicando el valor del identificador del elemento relativo.

- ***android:layout_above***: colocar el elemento encima del elemento referenciado.
- ***android:layout_below***: colocar el elemento debajo del elemento referenciado.
- ***android:layout_toLeftOf***: colocar el elemento a la izquierda del elemento referenciado.
- ***android:layout_toRightOf***: colocar el elemento a la derecha del elemento referenciado.
- ***android:layout_alignTop***: indica que el extremo superior de este elemento está alineado

con el extremo superior del elemento referenciado.

- ***android:layout_alignBottom***: indica que el extremo inferior de este elemento está alineado con el extremo inferior del elemento referenciado.
- ***android:layout_alignLeft***: indica que el extremo izquierdo de este elemento está alineado con el extremo izquierdo del elemento referenciado.
- ***android:layout_alignRight***: indica que el extremo derecho de este elemento está alineado con el extremo derecho del elemento referenciado.
- ***android:layout_alignBaseLine***: indica que las líneas de base de este elemento están alineadas con las del elemento referenciado.

A continuación se muestra un ejemplo de formulario de entrada de datos con este layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/nomEdit"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:hint="@string/email" />

    <EditText
        android:id="@+id/prenomEdit"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_below="@id/nomEdit"
        android:hint="@string/pass" />

    <Button
        android:id="@+id/valider"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@id/prenomEdit"
        android:layout_below="@id/prenomEdit"
        android:text="@string/ok" />

    <Button
        android:id="@+id/annuler"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@id/valider"
        android:layout_toLeftOf="@id/valider"
        android:text="@string/cancel" />

</RelativeLayout>
```

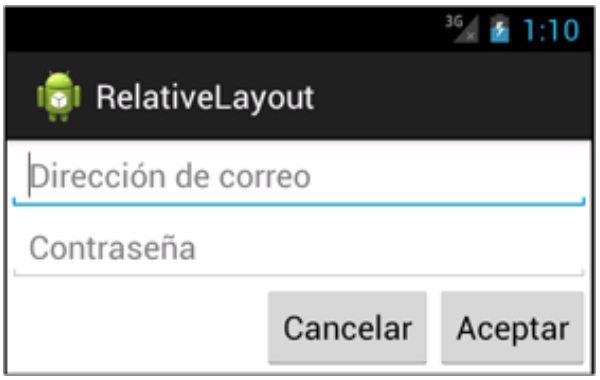
El primer campo está alineado con el extremo superior y derecho del contenedor.

El segundo campo de entrada está alineado por su extremo derecho con el extremo derecho del contenedor y por debajo del elemento anterior.

El primer botón se sitúa debajo del segundo campo, alineado a su derecha.

El segundo botón está alineado por arriba con el primer botón y se encuentra justo a su izquierda.

Con lo que obtenemos:



Puede combinar varios valores de posicionamiento.

También puede referirse a un elemento que se declarará posteriormente en la vista. Para ello, hay que declarar el identificador del siguiente elemento en la referencia (@+id) y reutilizarlo en la declaración del elemento.

Por ejemplo, se desea posicionar un campo de texto (**EditText**) tras un botón cuando se ha declarado antes en el archivo. Para poder asociar el campo de texto al botón y, en particular, a su identificador, hay que declarar el identificador del botón en la referencia del campo de texto (**android:layout_below**) y asignarla al botón posteriormente (**android:id**).

```
<EditText android:id="@+id/login"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/connect" />

<Button android:id="@id/connect"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/connect" />
```

Puede centrar elementos en un RelativeLayout mediante las opciones:

- **android:layout_centerHorizontal**: permite centrar un elemento horizontalmente.
- **android:layout_centerVertical**: permite centrar un elemento verticalmente.
- **android:layout_centerInParent**: permite centrar un elemento en el contenedor padre.

5. GridLayout

La versión Ice Cream Sandwich de Android trae consigo un gran surtido de novedades, una de las cuales es el GridLayout.

Este nuevo layout permite dividir la pantalla en filas y columnas, con la posibilidad de dejar celdas vacías o bien agrupar celdas de la misma fila o columna (lo que supone una deprecación del TableLayout).

El elemento **spacer** permite dejar una celda vacía y, de este modo, tener espacios vacíos en una interfaz gráfica.

A continuación se muestra una interfaz que contiene cuatro botones repartidos en dos filas.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<GridLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:columnCount="3" >

      <Space
          android:layout_width="wrap_content"
          android:layout_height="wrap_content" />

      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/btn5" />

      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/btn6" />

      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/btn7" />

      <Button
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:layout_columnSpan="2"
          android:text="@string/btn8" />

</GridLayout>
```

El atributo **columnCount** sirve para indicar que este **GridLayout** se compone de tres columnas.

La primera fila se compone de un **spacer** (espacio vacío) y de dos botones.

La segunda fila contiene dos botones, de los cuales el segundo se extiende ocupando dos columnas (atributo **layout_columnSpan**).

A continuación se muestra el resultado de la ejecución del ejemplo:

Recursos

La externalización de recursos es un elemento importante del desarrollo para Android (constantes, imágenes, animaciones, menús, vistas...). Esta externalización le permite mantener y actualizar los recursos más fácilmente.

El nombre de los archivos de recursos sólo puede contener letras minúsculas, cifras, un punto o un underscore (_).

Las carpetas que contienen recursos pueden tener varias características propias en función del idioma, del tamaño de la pantalla, del hardware, etc. Cada característica se separa mediante un guión (-).

A continuación se muestra una lista no exhaustiva de estas características:

- Nombre de código de país.
- Idioma.
- Anchura mínima de la pantalla: por ejemplo **sw<dimension>dp**.
- Anchura de la pantalla: por ejemplo **w<dimension>dp**.
- Altura de la pantalla: por ejemplo **h<dimension>dp**.
- Tamaño de la pantalla: **small**, **medium**, **large** o **xlarge**.
- Orientación de la pantalla, etc.
- Versión de Android (v14, v11, v8...).

1. Drawable

Esta carpeta se dedica, en gran parte, a la gestión de imágenes en una aplicación (véase el capítulo Personalización y gestión de eventos - Personalización).

a. Gestión de diferentes resoluciones

Para gestionar las distintas resoluciones de una imagen en una aplicación, dispone del grupo de carpetas **drawable-xxx** donde xxx puede reemplazarse por:

- **ldpi**: para pantallas con una resolución baja (alrededor de 120 dpi).
- **mdpi**: para pantallas con una resolución media (alrededor de 160 dpi).
- **hdpi**: para pantallas con una resolución alta (alrededor de 240 dpi).
- **xhdpi**: para pantallas con una resolución muy alta (alrededor de 320 dpi).
- **xxhdpi**: para pantallas con una resolución muy, muy alta (alrededor de 480 dpi).
- **nodpi**: contiene recursos que no dependen de la resolución de su dispositivo.

2. Values

Esta carpeta sirve para almacenar diferentes valores (constantes) y usarlos en la aplicación.

a. Cadenas de caracteres

Creación de cadenas de caracteres

Todas las cadenas de caracteres se almacenan en uno o varios archivos (**strings.xml** por defecto). La elección del nombre del archivo es completamente libre, pero es imprescindible que se encuentre en la carpeta **values**.

A continuación se muestra un ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="nombre_string">texto_string</string>
</resources>
```

Una vez se ha declarado una cadena de caracteres, ya se puede usar desde:

- Un archivo Java: R.string.nombre_string.
- Un archivo XML: @string/nombre_string.

El framework Android proporciona varios recursos nativos que puede usar en su aplicación. Si desea referenciar una cadena de caracteres proporcionada por el framework Android, utilice **@android:string/nombre_de_la_cadena**.

Gestión de plurales

Para permitirle gestionar el plural de sus cadenas de caracteres, Android integra una etiqueta muy práctica (**plurals**).

A continuación se muestra un pequeño ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="num_de_tutos">
        <item quantity="zero">No hay tutoriales
disponibles</item>
        <item quantity="one">Hay un tutorial disponible</item>
        <item quantity="other">Hay varios tutoriales
disponibles</item>
    </plurals>
</resources>
```

El atributo **quantity** (cantidad) puede tener los siguientes valores:

- **zero**: para el caso de cero elementos.
- **one**: para el caso de un elemento.
- **two**: para el caso de dos elementos.
- **few**: para el caso de pocos elementos (tres o cuatro, por ejemplo).
- **many**: para el caso de un gran número de elementos (diez o doce, por ejemplo).
- **other**: para el resto de casos.

Para poder obtener estas cadenas de caracteres en su código, está disponible el método ***getQuantityString***.

```
public String getQuantityString(int id, int quantity);
```

Este método recibe por parámetro:

- El identificador de la cadena deseada.

- La cantidad deseada.

A continuación se muestra un ejemplo para la declaración anterior:

```
Resources res = getResources();
String numberOfTutos =
res.getQuantityString(R.plurals.num_de_tutos,
cantidadDeseada);
```

El método **getResources** permite acceder a los recursos desde una actividad.

La llamada al método **getQuantityString** permite obtener la cadena deseada.

Escapar los apóstrofes

Para incluir apóstrofes en una cadena de caracteres, debe escaparlos como se indica a continuación:

```
<string name="apos1">"El ejemplo de un escape de apóstrofe (')"</string>
<string name="apos2">El ejemplo de un escape de apóstrofe (\')</string>
```

La primera forma de escapar apóstrofes en una cadena es incluyendo el texto entre comillas dobles.

La segunda manera de escapar apóstrofes es añadiéndoles una contrabarra (\) delante de cada uno de ellos.

Añadir argumentos

Puede añadir argumentos a sus cadenas de caracteres para adaptarlas mejor a las distintas situaciones que puedan darse en la aplicación.

A continuación se muestra un ejemplo:

```
<string name="reception_message">%1$s! Ha recibido %2$d
mensajes nuevos.</string>
```

La cadena de caracteres contiene dos argumentos:

- El primero corresponde con una cadena de caracteres (%s).
- El segundo con un valor entero (%d).

A continuación, en la aplicación, puede definir estos argumentos mediante el método **format**, que toma como argumento la cadena de caracteres y los parámetros que se añadirán a la misma.

```
Resources res = getResources();
String message =
String.format(res.getString(R.string.reception_message), name,
newMsgNumber);
```


Tenga cuidado con el orden de los argumentos.

¿Y HTML?

Puede usar algunas etiquetas HTML para personalizar las cadenas de caracteres.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
    <string name="bienvenido">Bienvenido a <b>tutos-
android.com</b>!</string>
</resources>
```

Las etiquetas soportadas son:

- b: negrita
- i: itálica
- u: subrayado

Puede crear otro archivo para almacenar los valores enteros y booleanos, con el mismo principio que para las cadenas de caracteres.

b. Tablas

Puede, fácilmente, almacenar tablas de cadenas de caracteres para los menús, listas o desplegables. Para ello, utilice el archivo arrays.xml en el que puede almacenar tablas de cadenas de caracteres.

A continuación se muestra un ejemplo de archivo arrays.xml:

```
<resources>
<string-array name="day">
   <item>Lunes</item>
   <item>Martes</item>
   <item>Miércoles</item>
   <item>Jueves</item>
   <item>Viernes</item>
   <item>Sábado</item>
   <item>Domingo</item>
</string-array>
</resources>
```

También puede declarar tablas de enteros, por ejemplo.

c. Dimensiones

Durante la creación de una aplicación Android, tendrá que definir varios valores para los tamaños o espacios de diferentes elementos. Para factorizar su código, puede crear el archivo dimens.xml, en el que almacenará y utilizará las diferentes dimensiones.

A continuación se muestra un ejemplo del contenido del archivo dimens.xml:

```
<resources>
   <dimen name="anchura_boton">60dp</dimen>
   <dimen name="altura_boton">35dp</dimen>
</resources>
```

Y un ejemplo de uso de estos valores en una interfaz:

```
<Button
        android:layout_width="@dimen/anchura_boton"
        android:layout_height="@dimen/altura_boton"
        android:text="@string/btn" />
```

El uso de dimensiones se realiza mediante una llamada al recurso "dimen" y especificando el

identificador del valor deseado.

Para declarar las dimensiones utilizadas en la aplicación, puede usar las siguientes unidades:

- px: píxeles - se corresponde con los píxeles reales de la pantalla.
- in: pulgadas - basado en el tamaño físico de la pantalla.
- pt: punto.
- mm: milímetro.
- dp: densidad de píxeles independientes - recomendado para mostrar elementos.
- sp: escala de píxeles independientes - recomendado para el tamaño de fuentes.

Este archivo también le permite gestionar las diferencias de tamaño de pantalla, creando distintas carpetas. Por ejemplo, puede crear un archivo dimens.xml en la carpeta values-large para gestionar las pantallas grandes.

d. Colores

Puede almacenar todos los colores que utilice su aplicación en un archivo de recursos para externalizarlos y utilizarlos más fácilmente.

```
<resources>
    <color name="red">#F00</color>
</resources>
```

La especificación de colores puede realizarse de varias formas:

- #RGB
- #RRGGBB
- #ARGB
- #AARRGGBB

R: Rojo, G: Verde, B: Azul, A: Transparente

Elementos imprescindibles

A continuación, se presentan los elementos que componen la base de la creación de interfaces en Android.

1. Etiqueta de texto

El elemento que sirve para mostrar un texto en su interfaz es un **TextView**.

A continuación se muestra un ejemplo de uso de un TextView:

```
<TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/text" />
```

La especificación del texto que se desea mostrar se realiza mediante el atributo **android:text**.

La etiqueta de texto tiene muchos atributos que puede usar en su aplicación (tamaño, color y fuente del texto, posicionamiento, número de filas, etc.).

2. Campo de edición de texto

El elemento llamado **EditText** permite al usuario introducir un texto mediante el teclado.

A continuación se muestra un ejemplo de uso:

```
   <EditText
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:hint="@string/editHint" />
```

El atributo ***android:hint*** permite indicar al usuario el tipo de texto esperado.

Puede mejorar la experiencia del usuario mostrando un teclado específico en función del tipo de campo. Esto es posible gracias al atributo ***android:inputType***.

Este atributo puede tomar los siguientes valores:

- **text** (valor por defecto): teclado normal.
- **textCapCharacters**: teclado todo en mayúsculas.
- **textCapWords**: primera letra automáticamente en mayúsculas.
- **textAutoCorrect**: activa la corrección automática.
- **textMultiLine**: texto en varias líneas.
- **textNoSuggestions**: sin sugerencias de corrección.
- **textUri**: permite introducir una URL web.
- **textEmailAddress**: dirección de correo electrónico.
- **textEmailSubject**: asunto de correo electrónico.
- **textShortMessage**: activa el acceso directo a smiley en el teclado.
- **textPersonName**: permite introducir el nombre de una persona (muestra speech to text en la parte inferior izquierda).

- **textPostalAddress**: permite introducir una dirección postal (muestra speech to text en la parte inferior izquierda del teclado).
- **textPassword**: entrada de una contraseña.
- **textVisiblePassword**: entrada de una contraseña visible.
- **number/numberSigned/numberDecimal/phone/datetime/date/time**: teclado numérico.

Esta lista no es exhaustiva.


Puede utilizar varios valores separándolos con |.

3. Botón

Usar el elemento **Button** es muy sencillo, como puede ver en el siguiente ejemplo:

```
<Button android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:text="@string/btn" />
```

4. Checkbox

El elemento **Checkbox** representa una simple casilla de selección, como las que se pueden activar en los formularios web:

```
<CheckBox android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:checked="true"
 android:text="@string/checkbox" />
```

Puede definir el estado inicial de una checkbox mediante el atributo ***android:checked*** (true si activa, false en caso contrario).

5. Imagen

Para añadir imágenes en una aplicación fácilmente, podemos utilizar el elemento **ImageView**.

```
<ImageView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher"
        android:contentDescription="@string/app_name"/>
```

El atributo **android:src** permite especificar la imagen que se desea mostrar.

El atributo **android:contentDescription** permite proporcionar una breve descripción de la imagen mostrada (utilizado para accesibilidad).

6. Gestión del clic

El clic es una acción indispensable en la gestión de la interacción del usuario con su aplicación.

La gestión del clic puede hacerse de dos formas distintas:

- Gestionar el clic en los botones, de forma separada.

- Hacer que su actividad implemente la interfaz ***onClickListener***.

Para ilustrar mejor este concepto, se muestra, a continuación, el ejemplo de una interfaz que dispone de dos botones.

Gestionar el clic en los botones de manera separada

```
Button btn1 = (Button) findViewById(R.id.btn1);
btn1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
       Log.v("ClickListener", "Interacción con el botón 1");
    }
});

Button btn2 = (Button) findViewById(R.id.btn2);
btn2.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
       Log.v("ClickListener", "Interacción con el botón 2");
    }
});
```

Se han declarado, en la vista, dos botones inicializados mediante el método ***findViewById***.

La llamada al método ***setOnClickListener*** permite añadir un listener al clic de los botones.

Este método recibe por parámetro un nuevo listener que permite sobrecargar el método ***onClick***. Este listener se crea mediante el constructor de la clase **onClickListener**.

Todo el tratamiento realizado en el clic debe implementarse en el método ***onClick***, que recibirá por parámetro la vista que ha recibido dicho clic.

La actividad implementa la interfaz onClickListener

```
public class ClickListenerMethod2Activity extends Activity
implements OnClickListener {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

            Button btn1 = (Button) findViewById(R.id.btn1);
            btn1.setOnClickListener(this);

            Button btn2 = (Button) findViewById(R.id.btn2);
            btn2.setOnClickListener(this);
   }

   @Override
   public void onClick(View v) {
   switch (v.getId()) {
   case R.id.btn1:
      Log.v("ClickListener", "Interacción con el botón 1");
      break;
    case R.id.btn2:
      Log.v("ClickListener", "Interacción con el botón 2");
      break;
    default:
       break;
    }
```

```
        }
}
```

El primer paso consiste en hacer que la actividad implemente la clase **onClickListener**, lo que nos permite sobrecargar el método **onClick**.

En este método, se comprueba cuál es el identificador del botón pulsado y, en función de su valor, se ejecuta la acción correspondiente.

El último paso consiste en obtener los dos botones y especificar que la instancia de nuestra clase es la gestora del evento clic.

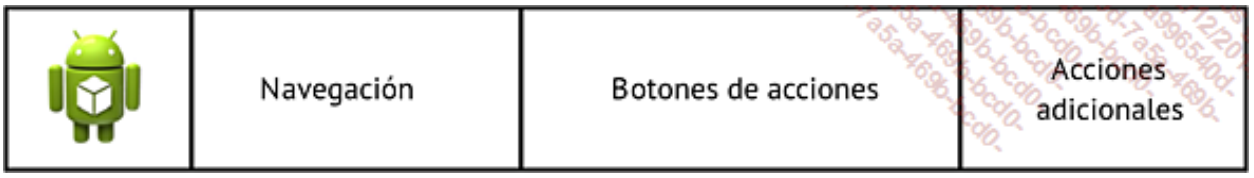
ActionBar

1. Principios

El concepto de ActionBar (barra de acciones) existe desde Android 3.0. Este elemento es una estructura importante de las aplicaciones Android. Se ubica en la parte superior de la pantalla de la aplicación y se muestra en todas las pantallas.

El objetivo de una ActionBar es:

- Hacer que las acciones más importantes de la aplicación sean fácilmente accesibles.
- Permitir navegar más fácilmente por la aplicación.



Una ActionBar se compone, generalmente, de varias secciones (de izquierda a derecha):

- **El icono de la aplicación**: permite establecer la identidad de una aplicación y volver a la pantalla anterior.
- **Navegación entre vistas**: este elemento permite al usuario navegar entre varias vistas. Esta navegación sirve, por ejemplo, para cambiar de cuenta o de carpeta en la aplicación Gmail.
- **Botones de acciones**: muestra las funcionalidades más importantes de una aplicación para un acceso rápido y simple.
- **Acciones adicionales**: este elemento permite acceder al resto de funcionalidades (menos importantes) de la aplicación.

También puede separar una ActionBar en tres partes según sus necesidades:

- Barra de acciones principales.
- Barra de tabulación (véase el capítulo Creación de interfaces avanzadas - Creación de pestañas).
- Barra de acciones secundaria.

Puede mostrar una barra de acciones contextual en función de las interacciones del usuario (por ejemplo, cuando el usuario selecciona un elemento de su lista).

Ejemplo

Las barras de acciones están disponibles para aplicaciones cuya versión del SDK objetivo sea la versión 11 como mínimo (3.0).

Por ejemplo:

```
<manifest ...>
    <uses-sdk android:minSdkVersion="15"
        android:targetSdkVersion="15" />
    ..........................
</manifest>
```

Para utilizar una ActionBar, el primer paso consiste en declarar un archivo xml en la carpeta **menu**.

Cada ítem representa un elemento que se incluirá en una ActionBar.

Un elemento tiene:

- Un identificador.
- Un icono.
- Un título.
- Un comportamiento en la ActionBar.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_search"
        android:icon="@drawable/ic_menu_search"
        android:showAsAction="ifRoom|withText"
        android:title="@string/menu_search"/>
    <item
        android:id="@+id/menu_exit"
        android:icon="@drawable/ic_menu_exit"
        android:showAsAction="ifRoom|withText"
        android:title="@string/menu_exit"/>
    <item
        android:id="@+id/menu_help"
        android:icon="@drawable/ic_menu_light"
        android:showAsAction="ifRoom|withText"
        android:title="@string/menu_light"/>
    <item
        android:id="@+id/menu_link"
        android:icon="@drawable/ic_menu_link"
        android:showAsAction="ifRoom|withText"
        android:title="@string/menu_link"/>

</menu>
```

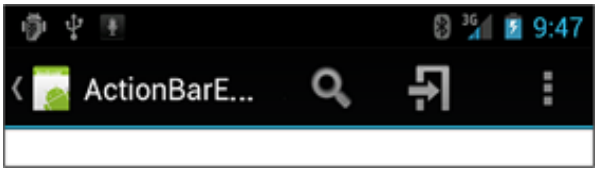
Puede observar el atributo ***android:showAsAction***, que permite especificar el comportamiento de un elemento perteneciente a una ActionBar. Puede adquirir los siguientes valores:

- **ifRoom**: ubicar el elemento en la ActionBar sólo si hay un hueco disponible (recomendado).
- **withText**: incluir el texto del elemento en la barra de acciones. Puede acoplar este valor con otros añadiendo el separador |.
- **never**: nunca ubicar en la barra de acciones.
- **always**: ubicar siempre el elemento en la barra de acciones. Se recomiendo no usar demasiado este valor salvo que se trate de una funcionalidad crítica, ya que si añade demasiados elementos a su barra de acciones, éstos pueden acabar superpuestos.
- **collapseActionView**: esta acción es retráctil (desde la API 14).

A continuación, sobrecargue el método ***onCreateOptionsMenu*** para asociar la barra de acciones a la actividad.

```
@Override
   public boolean onCreateOptionsMenu(Menu menu) {
          MenuInflater inflater = getMenuInflater();
          inflater.inflate(R.menu.actionbar, menu);
          return true;
   }
```

Con lo que obtendremos:



Puede observar el tercer icono, que representa tres puntos pequeños. Indica al usuario que hay otras acciones disponibles. Haciendo clic en estos tres puntos, el usuario podrá acceder al resto de funcionalidades de la aplicación.

Para gestionar el clic en los distintos elementos de una barra de acción, sobrecargue el método**onOptionsItemSelected** y realice la acción que corresponda al elemento del clic.

```
@Override
 public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
   case R.id.menu_exit:
       Log.v("ActionBarExempleActivity", "Clic en botón Salir");
       return true;
   case R.id.menu_help:
       Log.v("ActionBarExempleActivity", "Clic en botón Ayuda");
       return true;
   case R.id.menu_link:
       Log.v("ActionBarExempleActivity", "Clic en botón Enlaces");
       return true;
   case R.id.menu_search:
      Log.v("ActionBarExempleActivity", "Clic en botón Buscar");
      return true;
   }
   return super.onOptionsItemSelected(item);
}
```

Si desea que el clic en el icono de una ActionBar redirija al usuario a la pantalla anterior o a la pantalla principal de su interfaz, añada las siguientes líneas en el método ***onCreateOptionsMenu***:

```
ActionBar actionBar = getActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);
```

Puede crear barras de acciones en aquellas versiones de Android que no las soporten en la API nativa mediante múltiples librerías, como ***ActionBarSherlock***, o creando su propio componente.

Jelly Bean

Esta versión introduce un nuevo atributo (**android:parentActivityName**) que permite especificar la clase madre de una actividad (en el sentido de la navegación).

2. Separar la barra de acciones en dos

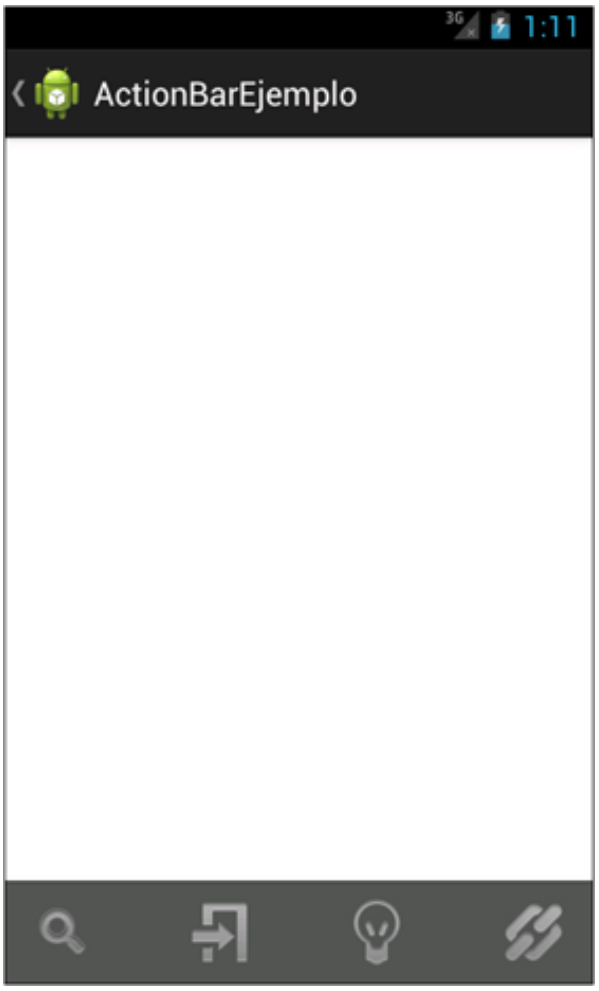
Esta funcionalidad sólo está disponible a partir de la API 14 (> Android 4.0). Permite separar una barra de acciones en dos. En este caso, aparece una segunda barra de acciones en la parte inferior de su aplicación.

Para activar esta opción, añada el atributo ***uiOptions="splitActionBarWhenNarrow"*** a las etiquetas **<activity>** (aplicar la separación para una actividad particular) o **<application>** (aplicar la separación para toda la aplicación) de su manifiesto.

Esta funcionalidad mejora enormemente la experiencia del usuario y le permite separar las barras de

acciones en función del tamaño de las pantallas.

Con lo que se obtendrá:



Puede utilizar algunas herramientas para generar el estilo de las barras de acciones:http://jgilfelt.github.com/android-actionbarstylegenerator/

Menús

Los menús siguen estando accesibles en la nueva versión de Android. Sin embargo, han pasado a un segundo plano debido a las bondades de las barras de acciones.

Si desea crear menús, es muy sencillo, y se hace de la misma forma que con las ActionBar. Puede crear menús para gestionar la compatibilidad hacia atrás de su aplicación con las versiones anteriores de Android que no tienen ActionBar.


Un menú tiene una apariencia diferente en función de la versión de Android en uso.

- Para comenzar, cree un archivo XML que represente su menú.

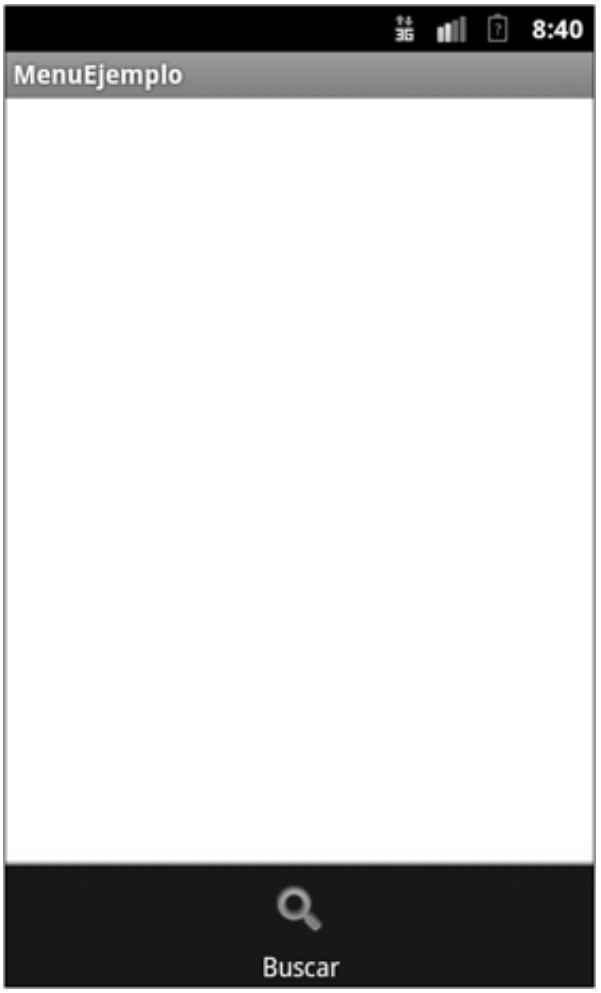
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_search"
        android:icon="@drawable/ic_menu_search"
        android:title="@string/menu_search"/>

</menu>
```

Puede observar que el archivo se parece al archivo XML que nos sirvió para declarar una ActionBar. La única diferencia es la ausencia del atributo **showAsAction**.

- A continuación, sobrecargue el método ***onCreateOptionMenu*** para cargar el archivo XML en la actividad.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
       MenuInflater inflater = getMenuInflater();
       inflater.inflate(R.menu.actionbar, menu);
       return true;
}
```

- Para finalizar, arranque el ejemplo y haga clic en el botón menú del emulador:



La gestión del clic en un menú se implementa del mismo modo que en barras de acciones.

Introducción

La navegación entre actividades en Android es posible gracias a los intents (véase el capítulo Principios de programación - Componentes Android). Cada aplicación se divide en varias actividades, de este modo cada actividad es accesible mediante un intent.

Gracias a los intents, puede implementar la lógica de su aplicación y, por lo tanto, facilitar su uso.

Esta aproximación permite establecer dos escenarios posibles:

- Se conoce la actividad que se debe iniciar.
- No se conoce la actividad que se debe iniciar, pero sí el tipo de acción que desea ejecutar.

Navegación entre pantallas

La navegación es un elemento esencial de la experiencia del usuario que proporciona la aplicación.

Para poder ejecutar una actividad, hay que utilizar el método **startActivity**.

```
public void startActivity (Intent intent)
```

Este método utiliza un intent que se corresponde con la actividad que desea ejecutar.

Para crear un intent, utilice el constructor proporcionado por la clase Intent. Este constructor recibe por parámetro un contexto y una referencia a la clase deseada.

```
Intent(Context packageContext, Class<?> activityToLaunch)
```

Si en una aplicación desea, en un momento dado, pasar de una actividad A a una actividad B después del clic en un botón, puede utilizar el siguiente código:

```
Intent intent = new Intent(A.this, B.class);
startActivity(intent);
```

No olvide declarar las nuevas actividades en el manifiesto de la aplicación.

Paso de datos entre pantallas

El paso de datos entre diferentes vistas de una aplicación se realiza con la ayuda de los extras (los intents los usan).

Un extra es un par clave/valor que utiliza el sistema de **Bundle**.

Gracias a los intents, va a poder pasar diferentes tipos de datos mediante el método **putExtra**. Este método está disponible para todos los tipos básicos de Java (int, string, float, double, byte, boolean, etc.).

Recibe dos parámetros:

- Una clave que identifica el elemento que se insertará.
- El valor del elemento.

Tomemos una pantalla de conexión que permita al usuario identificarse en la aplicación. A continuación, se necesitan pasar los datos de conexión a la siguiente pantalla.

La primera pantalla representa la interfaz de conexión, que se compone de dos campos de texto (usuario/contraseña) y de un botón que permite conectarse.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    >

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/user"
        android:id="@+id/user"/>

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/pass"
        android:id="@+id/pass"/>

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/connect"
        android:id="@+id/connect"/>

</LinearLayout>
```

Una segunda pantalla permite mostrar los datos obtenidos (dos etiquetas de texto).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    >

    <TextView
```

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/login"
        android:id="@+id/userLogin"/>

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/pass"
        android:id="@+id/userPassword"/>


</LinearLayout>
```

En la actividad correspondiente a la primera vista (pantalla de conexión), hay que:

- Obtener los diferentes elementos que componen la interfaz.
- Gestionar el clic en el botón de conexión (**onClickListener**).
- En el momento del clic, crear un intent que servirá para cambiar a la segunda actividad.

Este intent tendrá dos valores en extra (usuario y contraseña).

Estos valores se corresponden con el texto escrito en cada campo de texto.

```
    private final String EXTRA_LOGIN = "login";
    private final String EXTRA_PASSWORD = "password";

    private EditText login;
    private EditText password;
    private Button connection;

    @Override
     public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.main);

         login = (EditText) findViewById(R.id.user);
         password = (EditText) findViewById(R.id.pass);
         connection = (Button) findViewById(R.id.connect);
         connection.setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View v) {
                   Intent intent = new
Intent(Ch7_ScreenNavigationExempleActivity.this,
ConnectedActivity.class);
                  intent.putExtra(EXTRA_LOGIN,
login.getText().toString());
                  intent.putExtra(EXTRA_PASSWORD,
password.getText().toString());
                  startActivity(intent);
                      }
         });
      }
```

A continuación, hay que crear la segunda actividad que obtiene los datos y los muestra por pantalla.

La recuperación de los datos se realiza siguiendo los siguientes pasos:

- Obtención del intent de la actividad mediante el método **getIntent**.
- A continuación, se invoca al método que permite obtener el dato correspondiente al tipo de dato deseado. Por ejemplo, para las cadenas de caracteres, hay que utilizar el

método **getStringExtra**utilizando la misma clave que se usó en la inserción de los datos.

- Para finalizar, estos valores se asocian a los textos de los diferentes elementos.

```
private TextView login;
   private TextView password;

   private final String EXTRA_LOGIN = "login";
   private final String EXTRA_PASSWORD = "password";

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.connect);

      Intent intent = getIntent();
      String loginTxt = intent.getStringExtra(EXTRA_LOGIN);
      String passwordTxt =  intent.getStringExtra(EXTRA_PASSWORD);

      login = (TextView) findViewById(R.id.userLogin);
      login.setText(loginTxt);
      password = (TextView) findViewById(R.id.userPassword);
      password.setText(passwordTxt);
      }
```

No se olvide de declarar las actividades en el archivo de manifiesto de la aplicación.

```
<application
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name" >
   <activity
      android:name=".Cap7_ScreenNavigationEjemploActivity"
      android:label="@string/app_name" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
           <category
android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".ConnectedActivity"></activity>
</application>
```

Con lo que se obtendrán las siguientes dos pantallas:

1. Obtener un resultado

Puede ejecutar una actividad para realizar un tratamiento y recibir el resultado en la actividad origen. Por ejemplo, desde una actividad A, puede iniciar una actividad B para permitir al usuario que especifique unos datos y, a continuación, devolver el resultado a la actividad A.

- Para ello, debe ejecutar la actividad mediante el método **startActivityForResult(Intent, int)**:
  - El primer argumento se corresponde con la actividad que desea ejecutar.
  - El segundo argumento se corresponde con un código que permite identificar el resultado obtenido. Si este argumento es negativo, su llamada equivale a una llamada al método**startActivity**.
- A continuación, debe sobrecargar el método **onActivityResult(int requestCode, int resultCode, intent data)**. Este método tiene tres parámetros:
  - El primero permite identificar el origen del resultado (identificador usado en el método**startActivityForResult**).
  - El segundo permite especificar el tipo de resultado (KO/OK). Lo especifica la actividad invocada gracias al método **setResult** (RESULT_CANCELED o RESULT_OK).
  - La intención contiene los datos del resultado.

Puede crear sus propios valores de retorno ampliando los ya existentes por defecto (RESULT_OK, RESULT_CANCELED).

En la actividad que desea obtener el resultado, desencadene un intent hacia la segunda actividad.

```
Intent intent = new Intent(Cap7_ActivityResultEjemploActivity.this,
EligeResultadoActivity.class);
startActivityForResult(intent, RESULT_SELECTION);
```

En la segunda actividad, una vez se ha realizado el tratamiento, debe devolver el resultado a la actividad origen.

```
setResult(RESULT_OK);
finish();
```

El uso del botón retorno en el dispositivo, mientras se encuentra en la actividad invocada a través del método **startActivityForResult**, equivale a una llamada a**setResult(RESULT_CANCELED)**.

Puede especificar datos adicionales mediante un intent.

```
Intent intent = new Intent();
intent.putExtra("RESULT_OK", false);
setResult(RESULT_CANCELED, intent);
finish();
```

En la actividad principal, obtenga el resultado sobrecargando el método **onActivityResult**.

```
@Override  protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == RESULT_SELECTION) {
       if (resultCode == RESULT_OK) {
             Log.v("ActivityResult", "Result_OK");
       } else {
```

```
            Log.v("ActivityResult", "Result_KO");
        }
    }
}
```

Debe comprobar:

- Que el código del resultado se corresponde con el que desencadenó la llamada.
- El tipo del resultado y realizar los tratamientos adecuados.

2. Parcelable

Como se ha visto anteriormente, pasar datos entre actividades mediante el **Bundle** es muy fácil, pero este método tiene una limitación relativa a los tipos de datos que se pueden usar (tienen que ser datos nativos Java: Strings, int, Integer, boolean, etc.). Para superar esta limitación y transferir objetos personalizados, utilice los **parcelable**.

Android utiliza este sistema de forma nativa para transferir objetos entre actividades. Este sistema es parecido, en teoría, al sistema de **serialización** disponible en Java.

Para poder transferir un objeto personalizado de una actividad A a una actividad B, el objeto simplemente debe de implementar la interfaz **Parcelable**.

El siguiente ejemplo permite pasar los datos de conexión de un usuario entre dos actividades.

- Para empezar, cree una interfaz y una actividad que representen la vista de conexión del usuario.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/user"
        android:id="@+id/user"/>

      <EditText
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:hint="@string/pass"
            android:inputType="textPassword"
            android:id="@+id/pass"/>

      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/connect"
          android:id="@+id/connect"/>

</LinearLayout>
```

- A continuación, cree una segunda interfaz que permita mostrar los datos de conexión (usuario/contraseña).

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/user"
        android:id="@+id/userLogin"/>

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/pass"
        android:id="@+id/userPassword"/>
</LinearLayout>
```

Ahora, cree la clase que representa los datos de conexión de un usuario.

```
public class User {
   private String username;
   private String password;

   public User(String username, String password) {
          super();
          this.username = username;
          this.password = password;
   }
   public String getUsername() {
          return username;
   }

   public void setUsername(String username) {
          this.username = username;
   }

   public String getPassword() {
          return password;
   }

   public void setPassword(String password) {
          this.password = password;
   }
}
```

Como se ha dicho anteriormente, la clase que representa un usuario debe implementar la interfaz**Parcelable**.

Esta interfaz añade los siguientes dos métodos:

- ***describeContents***: sirve para describir el contenido del Parcel y, en particular, el número de objetos especiales (no nativos) alojados en el Parcel.
- ***writeToParcel***: sirve para escribir el contenido del objeto en el Parcel.

```
@Override public int describeContents() {
     return 0;
}
@Override public void writeToParcel(Parcel dest, int flags) {
     dest.writeString(username);
     dest.writeString(password);
```

```
}
```

El método **describeContents** devuelve 0, ya que nuestro objeto no contiene ningún objeto especial.

El método **writeToParcel** escribe dos cadenas de caracteres que representan los datos de usuario en el parcel. El flag permite describir cómo se debe escribir el objeto, puede adquirir los valores 0 o**PARCELABLE_WRITE_RETURN_VALUE**.

La siguiente etapa se corresponde con la creación de un objeto CREATOR que permita crear una instancia de nuestro objeto usuario desde un Parcel.

```
public static final Parcelable.Creator<User> CREATOR =  new
Parcelable.Creator<User>() {

    @Override
    public User createFromParcel(Parcel source) {
         return new User(source);
    }

    @Override
    public User[] newArray(int size) {
         return new User[size];
    }
};
```

Este objeto tiene algunas características:

- El objeto debe ser **public static**.
- El tipo del objeto debe parametrizarse con el nombre de la clase que implementa **Parcelable**.
- El método **createFromParcel** permite crear un objeto User desde un **Parcel**. Por lo tanto, hay que crear un constructor que reciba por parámetro un **Parcel** y devuelva un usuario.
- El método **newArray** permite crear una tabla de usuarios cuyo tamaño se especifica por parámetro.
- El constructor que sirve para crear un objeto User desde un **Parcel** es muy sencillo, basta con leer el contenido del **Parcel** y asignar los valores a una nueva instancia de la clase User.

Los valores deben leerse en el mismo orden que el que se ha definido en el método**writeToParcel**.

```
public User(Parcel source) {
   this.username = source.readString();
   this.password = source.readString();
}
```

Para pasar los datos de la actividad A a la actividad B, hay que crear el intent que servirá para ejecutar la actividad B y una instancia de la clase User para, después, simplemente utilizar el método **putExtra**.

```
Intent intent = new Intent(A.this, B.class);
User user = new User(login.getText().toString(),
password.getText().toString());
intent.putExtra(EXTRA_USER, user);
startActivity(intent);
```

Del lado de la clase invocada, hay que utilizar el método **getParcelableExtra** para obtener la instancia del usuario.

```
Intent intent = getIntent();
User user = intent.getParcelableExtra(EXTRA_USER);

login = (TextView) findViewById(R.id.userLogin);
login.setText(user.getUsername());
password = (TextView) findViewById(R.id.userPassword);
password.setText(user.getPassword());
```

Con lo que obtendremos:

Llamar a otras aplicaciones

Android permite utilizar otras aplicaciones disponibles en el dispositivo para realizar un tratamiento.

Por ejemplo, puede llamar a la aplicación de ajustes de Android para solicitar al usuario la activación de su GPS.

La forma más sencilla de llamar a otra aplicación es usando el **PackageManager** de Android.


Este método requiere que conozca el nombre del package de la aplicación invocada.

En el siguiente ejemplo, la aplicación de ajustes deberá abrirse cuando se haga clic en un botón de la vista.

```
Button launchSettings = (Button) findViewById(R.id.launchMap);
launchSettings.setOnClickListener(new OnClickListener() {

   @Override
   public void onClick(View v) {
      PackageManager pm = getPackageManager();
      Intent intent =
pm.getLaunchIntentForPackage("com.android.settings");
      if (intent != null)
            startActivity(intent);
      else
            Log.e("LaunchSettings",
      "La aplicación no está disponible en este dispositivo");
   }
});
```

Para comenzar, configure un listener para el clic en el botón y, a continuación, obtenga una instancia del **PackageManager** utilizando el método **getPackageManager**.

Para poder mostrar la actividad de ajustes, hay que obtener un Intent mediante el método**getLaunchIntentForPackage** especificando el nombre del package de la aplicación invocada.

Si el valor del intent obtenido es null, significa que la aplicación no está disponible en el dispositivo. En caso contrario, puede ejecutar la aplicación de ajustes.

Buscar una aplicación que permita ejecutar una acción

El sistema Android le puede proponer una lista de aplicaciones en función del tipo de acción que desee ejecutar.

Por ejemplo, en una aplicación, puede necesitar abrir un archivo PDF y, por lo tanto, utilizar una aplicación cuyo nombre e identificador del package desconoce. Para ello, hay que utilizar otra característica especial de los Intents.

```
Uri papth = Uri.fromFile(file);
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setDataAndType(path, "application/pdf");

PackageManager pm = getPackageManager();
ComponentName component = intent.resolveActivity(pm);

if (component == null) {
   Log.e("PDFIntentLaunch"  ,
   "No hay ninguna aplicación para abrir un archivo PDF");
} else {
   startActivity(intent);
}
```

Debe crear un intent que tenga las propiedades siguientes:

- **ACTION_VIEW**: permite mostrar datos a un usuario para que pueda validad una selección.

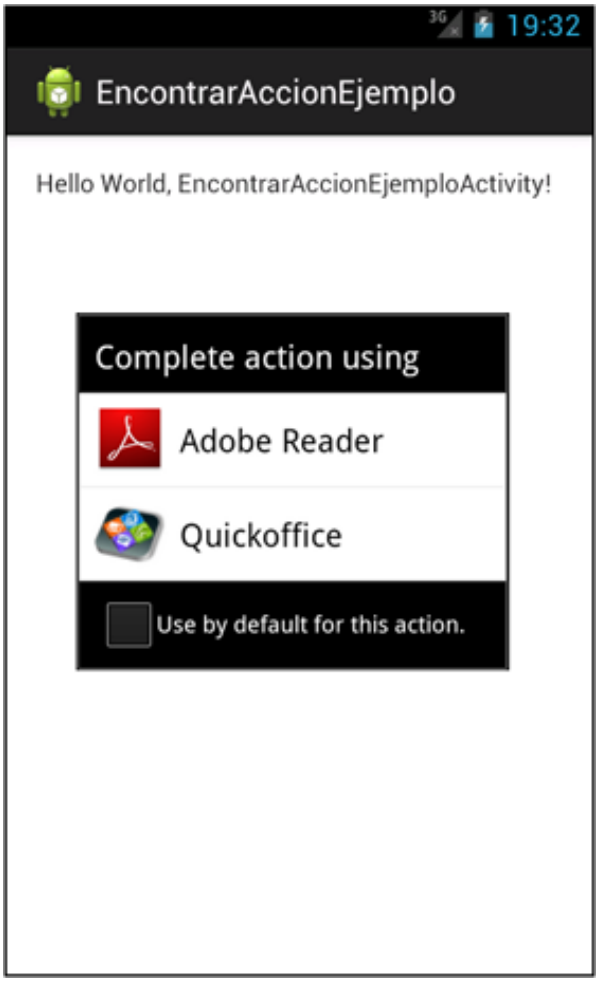
Debe especificar el tipo de datos así como el archivo que desea abrir (método **setDataAndType**).

Para saber si hay, al menos, una aplicación disponible para su intent en el dispositivo utilice el método**resolveActivity**.

Si el método **resolveActivity** devuelve el valor null significa que no hay ninguna aplicación disponible. Puede mostrar un mensaje de error adecuado o redirigir al usuario a una aplicación disponible en el market, por ejemplo.

En caso contrario, inicie la actividad.

Con lo que obtendrá:


19:32
EncontrarAccionEjemplo
Hello World, EncontrarAccionEjemploActivity!
Complete action using
Adobe Reader
Quickoffice
Use by default for this action.

Personalización

Uno de los puntos fuertes de Android reside en su presencia en un gran abanico de dispositivos (teléfonos, tablets, etc.). Este hecho aumenta el número potencial de usuarios, de dispositivos y de países que pueden acceder a una aplicación, pero también aumenta la fragmentación entre diferentes dispositivos (tamaño de pantalla, densidad y composición del hardware de los dispositivos, versión de Android, etc.).

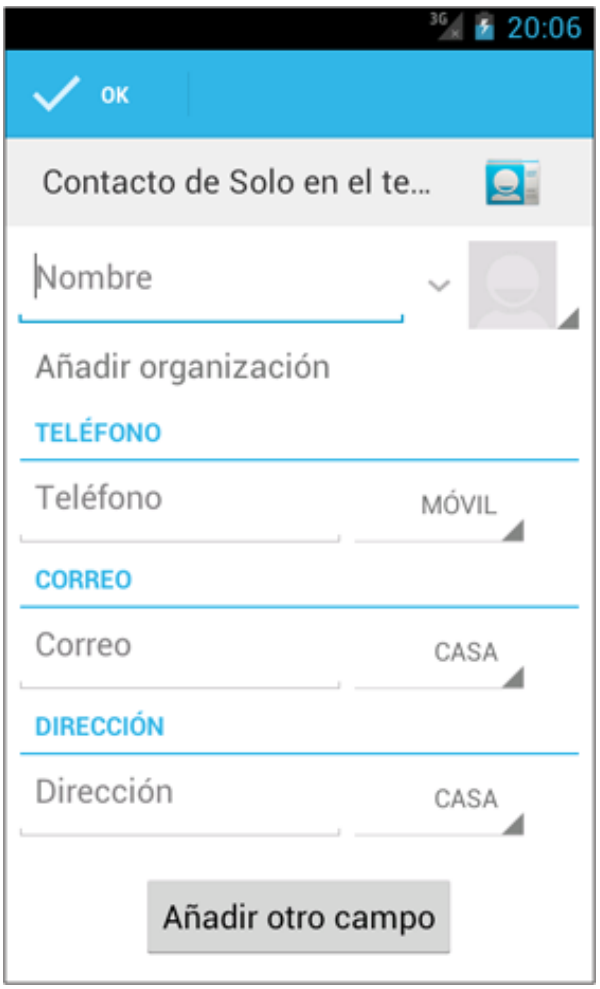
1. Temas

a. Definición

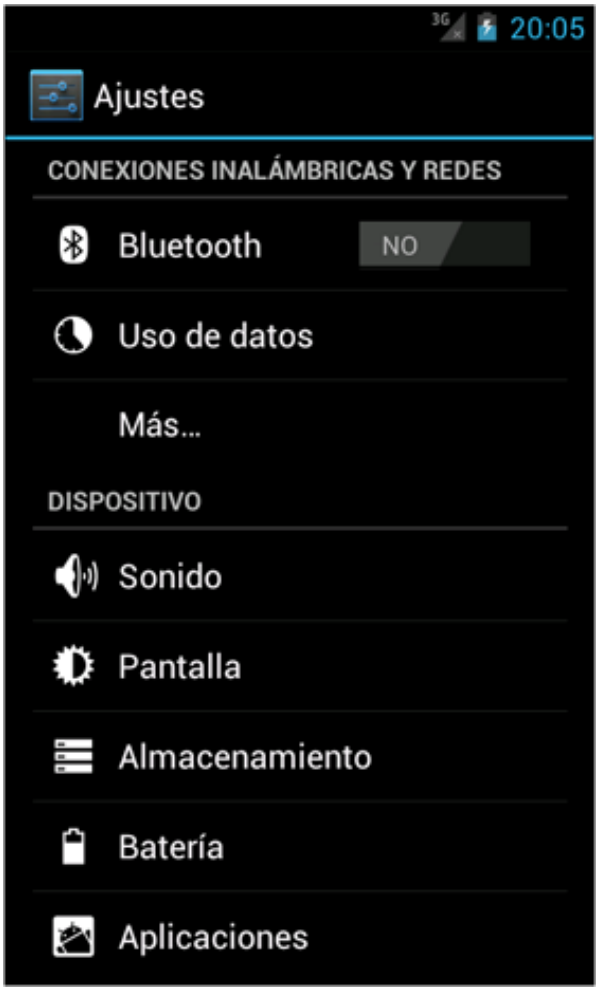
La personalización de una aplicación Android requiere, generalmente, el uso de temas. Un tema (un estilo) especifica el aspecto visual de los diferentes elementos que componen una aplicación definiendo sus propiedades visuales (color, tamaño, márgenes internos, tamaños de fuente, etc.).

Para conseguir que las distintas aplicaciones sean coherentes entre ellas y con el tema general del sistema, Android proporciona tres temas que puede utilizar o sobrecargar en una aplicación:

- Holo Light.
- Holo Dark.
- Holo Light con una ActionBar negra.



Holo Light



Holo Dark

b. Implementación

Utilizar el tema Android

Puede utilizar uno de los temas Holo de Android directamente en una aplicación. Para ello, especifique en la etiqueta **application** del manifiesto el tema que desea usar.

```
<application
     android:icon="@drawable/ic_launcher"
     android:label="@string/app_name"
     android:theme="@android:style/Theme.Holo">
```

Sobrecargar el tema Android

Puede sobrecargar el tema Android creando un archivo styles.xml en la carpeta values.

A continuación, cree la etiqueta **style** y, mediante el atributo **parent**, especifique el tema que va a sobrecargar.

```
<resources>
    <style name="MiTema" parent="@android:style/Theme.Holo">
       <!-- Personalizar el tema -->
           <item name="android:textColor">#F00</item>
    </style>
</resources>
```

Este método le permite guardar los valores de los atributos del tema deseado y especificar los atributos que quiere cambiar.

Puede crear un tema sin sobrecargar otro (para ello, quite el atributo parent).

2. Estado de los componentes

a. Estados

Cada componente de interfaz que forma parte de su aplicación puede tener varios estados:

- **Normal**: representa el estado general de un componente con ninguna interacción por parte del usuario.
- **Presionado**: estado que representa el componente siendo apretado.
- **Seleccionado**: representa el estado del componente una vez se ha seleccionado.
- **Desactivado**.
- **Seleccionado pero desactivado**.

b. Implementación

Para implementar los diferentes estados de un botón es necesario crear un archivo XML en el que se especifique el comportamiento de un botón en sus distintos estados. El elemento XML utilizado para especificar este comportamiento es la etiqueta **selector**.

Cree un archivo XML en su carpeta **drawable**.

```
<?xml version="1.0" encoding="utf-8"?>
<selector
xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:state_pressed="true" android:color="#F00"/>
    <item android:color="#000"/>

</selector>
```

Cada ítem se corresponde con un estado del botón.

Puede especificar el estado deseado mediante los siguientes atributos:

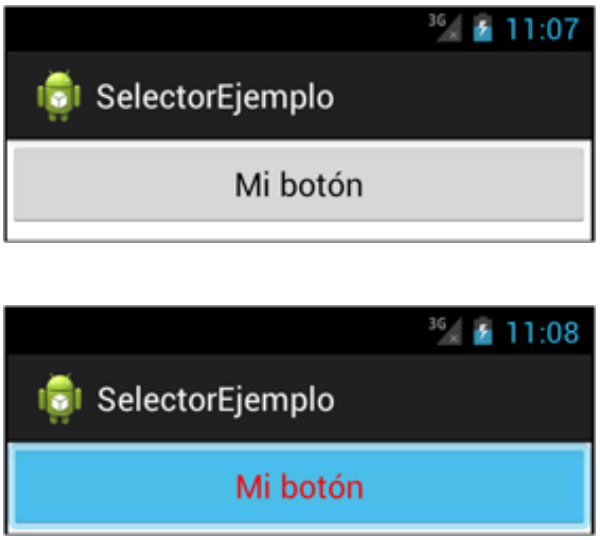
- **android:state_pressed**: se corresponde con el clic en un elemento.
- **android:state_focused**: se corresponde con un elemento con foco.
- **android:state_hovered**: se corresponde con un elemento sobrevolado.
- **android:state_selected**: se corresponde con la selección de un elemento.
- **android:state_checkable**: ocurre cuando el elemento puede ser activado.
- **android:state_checked**: ocurre cuando el elemento está activado.
- **android:state_enabled**: se utiliza cuando el elemento está habilitado.

Se han definido dos estados para el botón:

- El primer estado se corresponde con el momento en que el usuario lo pulsa. El color del texto del botón pasará a ser rojo.

- El segundo estado se corresponde con el estado normal del botón (color blanco).

Con lo que se obtendrán dos estados:



Los selectores sólo se aplican a elementos drawable (imagen o color de fondo de un componente, por ejemplo).

3. Gradiente

Puede crear un gradiente de manera muy sencilla en Android para usarlo en la aplicación (como color de fondo por ejemplo), utilizando una combinación de las etiquetas **shape** y **gradient**.

La etiqueta **shape** permite crear formas (rectángulo, círculo, línea...).

```
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape=["rectangle" | "oval" | "line" | "ring"] >
```

La etiqueta **gradient** permite crear gradientes siguiendo la forma definida por la etiqueta **shape**.

```
<gradient
        android:angle="integer"
        android:centerX="integer"
        android:centerY="integer"
        android:centerColor="integer"
        android:endColor="color"
        android:gradientRadius="integer"
        android:startColor="color"
        android:type=["linear" | "radial" | "sweep"]
 />
```

Dispone de los siguientes atributos:

- **android:angle**: representa el ángulo del degradado. Los valores deben ser un múltiplo de 45 (0 se corresponde con un gradiente de izquierda a derecha y 90 con uno de abajo a arriba).
- **android:centerX** et **android:centerY**: representa el centro del gradiente (X/Y). Este valor está comprendido entre 0 y 1.0.
- **android:centerColor**: representa el color del centro del gradiente.
- **android:startColor**: representa el color de inicio del gradiente.

- **android:endColor**: representa el color de final del gradiente.
- **android:gradiantRadius**: representa el radio del gradiente.
- **android:type**: representa el tipo de gradiente.

El siguiente ejemplo representa un gradiente que se asignará como color de fondo de una actividad (este archivo se deberá crear en la carpeta **Drawable**).

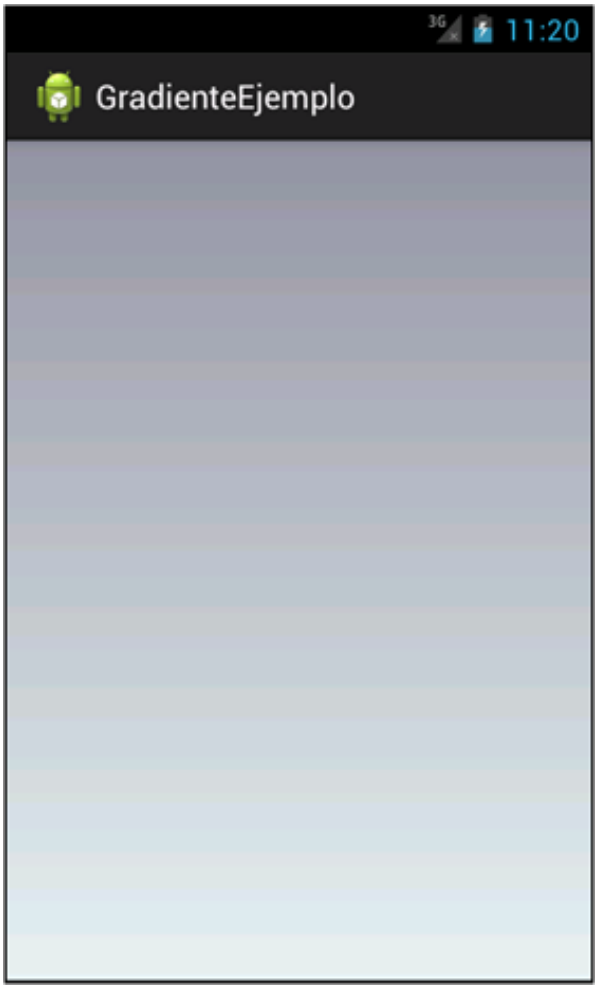
```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >

    <gradient
        android:angle="90"
        android:endColor="#33f240"
        android:startColor="#59de8e" />

</shape>
```

Este gradiente tiene una forma rectangular y un ángulo de 90 (de abajo a arriba).

Para poder asignar este gradiente como color de fondo de la actividad, hay que utilizar la etiqueta**android:background** y pasar como argumento el archivo que representa el gradiente.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:background="@drawable/gradient" >

</LinearLayout>
```

Con lo que se obtendrá:



4. Fuentes

Las nuevas versiones de Android (ICS y Jelly Bean) introducen la fuente **Roboto**, disponible en versión normal y en negrita.

La gestión de las características de una fuente en Android se realiza mediante los atributos clásicos (espaciado, escala, alineación, tamaño, etc.).

Tamaño de fuente

Puede definir el tamaño de los textos de su aplicación mediante el atributo **android:textSize**. Sin embargo, se recomienda utilizar ciertos valores.

- Texto muy pequeño: 14sp.
- Texto pequeño: 18sp.
- Texto normal: 22sp.
- Texto grande: 26sp.

Cuando defina el tamaño del texto, utilice las medidas en sp (Scale Independent Pixel).

Tipo de fuente

Para especificar el tipo de fuente que desea utilizar en los textos de una aplicación, utilice el atributo***android:typeface***.

Este atributo puede tener los siguientes valores:

- **DEFAULT**: tipo de fuente por defecto.
- **DEFAULT_BOLD**: tipo de fuente por defecto, pero en negrita.
- **MONOSPACE**: tipo de fuente monospace.
- **SANS_SERIF**: tipo de fuente sans serif.
- **SERIF**: tipo de fuente serif.

Estilo de fuente

Puede definir el estilo de una fuente mediante el atributo **android:textStyle**. Los estilos que permite Android son:

- Normal.
- Negrita.
- Cursiva.

Fuente personalizada

También puede usar una fuente personalizada para una aplicación. Para ello, debe incluir la fuente dentro del apk de la aplicación.

Para importar una fuente, ubique el archivo .ttf que representa la fuente en la carpeta **assets** del proyecto Android.

A continuación se muestra un ejemplo que representa un campo de texto que tiene una fuente personalizada.

Una vez se ha creado el campo de texto en un archivo XML, modifique la actividad y agregue la porción de código que permite cargar la fuente personalizada.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```

```
    Typeface font = Typeface.createFromAsset(getAssets(),
"miFuente.ttf");
    TextView miFuente = (TextView)
findViewById(R.id.miFuente);
   miFuente.setTypeface(font);
}
```

Se ha creado una instancia de la clase **Typeface** a partir del archivo **ttf** ubicado en la carpeta **assets**. El método **createFromAsset** permite crear un nuevo **Typeface** de forma sencilla a partir del archivo especificado como parámetro.

Para finalizar, defina el **Typeface** del campo de texto declarado anteriormente.

Con lo que se obtendrá:

5. Iconos

En una aplicación, dispone de varios tipos de iconos:

- El icono del Launcher: incluido en el lanzador de la aplicación y/o en el escritorio.
- Los iconos presentes en la ActionBar.
- Los iconos de notificación.
- El resto de iconos (en las listas, por ejemplo).

a. Icono del Launcher

Este icono representa la identidad visual de una aplicación. Se debería reconocer fácilmente una aplicación a partir de su icono.

A continuación se muestran las propiedades de un buen icono de launcher. Éste debe ser:

- Visible y claro sea cual sea el fondo de pantalla del teléfono.
- Sencillo y no sobrecargado de elementos.
- Disponible en todas las resoluciones.

b. Iconos de la ActionBar

Los iconos que se muestran en la barra de acciones deben ser simples, representar claramente la acción asociada y tener, por lo general, un color grisáceo.

Puede descargar un pack de iconos publicados por Google en la dirección siguiente:https://dl-ssl.google.com/android/design/Android_Design_Icons_20120229.zip

Animaciones

Las animaciones representan efectos visuales que pueden aplicarse a varios elementos de la interfaz. Android tiene dos tipos de animaciones:

- El primer tipo afecta a las propiedades de las vistas (tamaño, posición, opacidad). A este tipo de animación se le denomina **Tween Animation**.
- El segundo tipo permite mostrar, entre otras, varias imágenes en una misma vista. Este tipo de animación se llama **Frame animation**.

1. Tween Animation

Esta animación permite realizar transiciones (rotación, opacidad, movimiento). Se define en un archivo XML en la carpeta ***anim***, dentro de la carpeta ***res***.

El archivo XML que representa este tipo de animaciones puede contener las etiquetas siguientes:

- ***set***:
  - El archivo debe, obligatoriamente, comenzar con una etiqueta set.
  - Cada etiqueta representa un grupo de animación.
  - Puede incluir varias etiquetas set para crear varios grupos de animaciones.
- ***alpha***: la animación utiliza transparencia de elementos.
- ***scale***: la animación sirve para redimensionar elementos. Puede especificar el centro usado para dimensionar el elemento.
- ***translate***: representa una translación horizontal o vertical del elemento deseado.
- ***rotate***: la animación sirve para realizar una rotación.

Para ilustrar estas explicaciones, se va a crear una animación que sirva para realizar la rotación de una imagen.

Cree, para empezar, una vista que incluya una imagen (**ImageView**):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ImageView
      android:id="@+id/image"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical|center_horizontal"
      android:src="@drawable/android" />

</LinearLayout>
```

A continuación, en la carpeta ***anim***, cree un archivo llamado ***my_animation.xml*** en el que definirá su animación.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:shareInterpolator="false" >

    <rotate
```

```
        android:duration="2000"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:repeatCount="3"
        android:toDegrees="360" />

</set>
```

La animación representa una rotación que tiene las propiedades siguientes:

- ***android:duration***: duración de la rotación (en milisegundos).
- ***android:fromDegress***: ángulo de inicio de la rotación.
- ***android:pivotX*** y ***android:pivotY***: sirven para definir el centro de la rotación.
- ***android:repeatCount***: número de repeticiones de la rotación (3 veces).
- ***android:toDegrees***: ángulo de fin de la rotación.

Después, para finalizar, cargue la animación para asociarla al elemento deseado:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ImageView image = (ImageView) findViewById(R.id.image);
    Animation myAnimation = AnimationUtils.loadAnimation(this,
R.anim.my_animation);
    image.startAnimation(myAnimation);
}
```

Una vez se ha obtenido la instancia de la imagen, se debe cargar la animación mediante el método**loadAnimation**. Este método recibe como parámetros el contexto y la animación que se desea cargar.

Para terminar, hay que usar el método **startAnimation** para iniciar la animación.

El resultado obtenido corresponde al de una imagen que realiza una rotación de 360 grados 3 veces seguidas. Cada rotación dura 2000 milisegundos.

2. Frame Animation

Esta animación permite mostrar una sucesión de imágenes en un orden predefinido. Para comenzar, cree una interfaz compuesta por una imagen.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ImageView
        android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

A continuación, cree el archivo que representa la animación en la carpeta **anim**(frame_animation.xml).

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/market1"
android:duration="400" />
    <item android:drawable="@drawable/market2"
android:duration="500" />
    <item android:drawable="@drawable/market3"
android:duration="600" />
</animation-list>
```

Este archivo se compone de una lista de elementos. Cada uno de ellos representa una imagen. Para cada elemento, debe especificar la imagen así como la duración de la visualización (en milisegundos).

El atributo **android:oneshot** significa que la animación sólo se ejecutará una vez. En el ejemplo, se ejecutará varias veces debido a que el atributo está definido a falso.

Para terminar, hay que asociar la imagen a la animación en el archivo correspondiente a la actividad.

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);

   ImageView marketImage = (ImageView) findViewById(R.id.image);
   marketImage.setBackgroundResource(R.anim.frame_animation);

   final AnimationDrawable marketAnimation = (AnimationDrawable)
marketImage.getBackground();
   marketImage.post(new Runnable() {
      public void run() {
         if ( marketAnimation != null )
             marketAnimation.start();
      }
   });
}
```

La animación se aplica a la imagen de fondo de la vista, por lo que hay que asociar el background de la imagen a la animación. Después, hay que obtener una instancia de la clase **AnimationDrawable**para iniciar la animación en un nuevo thread usando el método **start**.

Es obligatorio ejecutar la animación en otro thread para no bloquear el thread del UI.

El resultado obtenido corresponde a una sucesión de tres imágenes que representan la evolución de los iconos de Google Play desde su creación.

3. Cambio de vista

Puede sobrecargar la animación que se utiliza en el paso de una vista a otra. Puede diferenciar dos categorías:

- Animación de apertura de actividad.
- Animación de cierre de actividad.

Para comenzar, cree un archivo llamado **styles.xml** en la carpeta **values**. En este archivo, cree un primer tema al que llamará **customTheme**.

```
<style name="customTheme" parent="@android:style/Theme.Holo">
      <item name="android:windowAnimationStyle">
                @style/ActivityAnimation</item>
</style>
```

Este tema personalizado sobrecarga el tema Holo de Android modificando el atributo utilizado para la animación de las actividades. Este atributo tendrá por valor el tema creado en el siguiente paso y que permitirá especificar la animación utilizada para la apertura de una actividad (**activityOpenEnterAnimation**).

```
<style name="ActivityAnimation"
parent="@android:style/Animation.Activity">
    <item name="android:activityOpenEnterAnimation">@anim/
start_activity</item>
    </style>
```

Este atributo hace referencia a una animación creada en la carpeta **anim**. Esta animación realiza una simple rotación de la actividad.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
     <rotate
        android:duration="2000"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:toDegrees="360" />
</set>
```

El último paso permite asociar el **customTheme** a la aplicación. Para ello, hay que modificar el archivo de manifiesto para actualizar el tema usado (atributo **android:theme**).

```
<application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/customTheme">
```

El resultado obtenido permite realizar una rotación de la nueva actividad antes de mostrarla.

También puede crear animaciones dinámicamente con varias clases: **ViewPropertyAnimator, LayoutTransition**, etc.

Internacionalización

Una de las características específicas del market Android es la posibilidad de hacer que una aplicación esté disponible a nivel internacional. Para permitir el acceso y el uso de la aplicación a un amplio abanico de usuarios, deberá gestionar el mayor número de idiomas posible en sus aplicaciones.

Cuando se arranca una aplicación en un teléfono, el idioma utilizado por la aplicación se corresponde con el del teléfono si la aplicación soporta ese idioma, en caso contrario la aplicación mostrará su idioma por defecto.

La traducción de cadenas de caracteres es el elemento esencial para internacionalizar una aplicación. Sin embargo, no olvide proporcionar versiones de sus imágenes, sonidos y vídeos adaptados a las distintas localizaciones.

La carpeta **values**, ubicada en la estructura de carpetas del proyecto, se utiliza para dar soporte al idioma por defecto de una aplicación. Este idioma se aplicará en la aplicación si el idioma del usuario no está soportado.

Utilice la carpeta **values** para almacenar los archivos en inglés (idioma por defecto recomendado).

Tomemos el ejemplo de una aplicación que desea gestionar los idiomas español, inglés, francés y japonés.

- La carpeta **values**, incluida por defecto en la aplicación, contendrá los archivos en el idioma inglés.
- Para el resto de idiomas, hay que crear las siguientes carpetas:
  - Francés: **values-fr**
  - Español: **values-es**
  - Japonés: **values-ja**

Por lo tanto, deberá crear una carpeta por idioma que soporte la aplicación y proporcionar los archivos traducidos en los idiomas deseados.

Gestión de eventos

1. Pulsación de teclas

Para gestionar los eventos de pulsación sobre una tecla del teléfono dispone de dos métodos:

- **onKeyDown**: evento producido cuando se pulsa una tecla.
- **onKeyUp**: evento producido cuando se suelta una tecla.
- Ambos métodos tienen el mismo prototipo y reciben dos parámetros:
  - El primer parámetro es un valor entero que se corresponde con la tecla del teléfono del evento.
  - El segundo parámetro es una descripción del evento.

El método debe devolver verdadero si ha recuperado y tratado el evento y falso en caso contrario, propagando de este modo el evento.

A continuación se muestra un ejemplo de implementación del método **onKeyDown** para obtener la interacción del usuario con las teclas de volumen del teléfono.

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
     super.onKeyDown(keyCode, event);
     if (keyCode == KeyEvent.KEYCODE_VOLUME_DOWN) {
            Log.v("KeyEventActivity", "Ha pulsado la tecla
VOLUME DOWN");
          return true;
   }
     if (keyCode == KeyEvent.KEYCODE_VOLUME_UP) {
             Log.v("KeyEventActivity", "Ha pulsado la tecla
VOLUME UP");
          return true;
   }
          return false;
}
```

En este ejemplo, se comprueba el tipo de tecla pulsada. Si se corresponde con una de las dos teclas de volumen, se muestra el Log adecuado.

Puede sobrecargar los métodos **onTrackballEvent** y **onTouchEvent** para gestionar, respectivamente, los eventos trackball y la interacción del usuario con la pantalla del dispositivo.

2. Supervisión de la entrada

En algunas aplicaciones, por ejemplo, un cliente de Twitter, necesitará supervisar el estado de la entrada de un texto en un campo de texto. Esta supervisión se puede implementar usando la clase**TextWatcher**.

En este ejemplo se puede indicar al usuario el número de caracteres restantes que puede escribir o los que ha escrito en exceso.

Para comenzar, cree un nuevo proyecto Android y, a continuación, abra el archivo correspondiente a la vista principal (**main.xml**).

Este archivo se debe componer de un campo de texto (**EditText**), de una etiqueta (**TextView**) y de un botón (**Button**). Cada elemento tiene un identificador único.

- El objetivo es supervisar el campo de texto (limitado a 20 caracteres).
- La etiqueta permite indicar al usuario el número de caracteres restantes o los que ha introducido de más.
- El botón permite enviar el mensaje y se desactiva cuando el usuario supera el número de caracteres máximo.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/msg"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/msg" />

    <TextView
        android:id="@+id/indicator"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/numChar"
        android:textSize="25sp" />

    <Button
        android:id="@+id/send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/sendMessage" />

</LinearLayout>
```

La actividad correspondiente a nuestra vista deberá implementar la clase **TextWatcher** y, por lo tanto, sobrecargar los siguientes tres métodos:

- **afterTextChanged(Editable s)**: este método se invoca para notificar que el texto supervisado ha cambiado.
- **beforeTextChanged(CharSequence s, int start, int count, int after)**: este método se invoca para notificar que el texto supervisado está a punto de cambiar. Puede permitirle, por ejemplo, reaccionar a los cambios que se realizarán al texto.
- **onTextChanged(CharSequence s, int start, int count, int after)**: este método se invoca cuando se han realizado cambios al texto supervisado.

```
public class Cap7_TextWatcherExempleActivity extends Activity
implements TextWatcher {

   private EditText msg;
   private TextView textIndicator;
   private Button send;

   private final static int NUMMAXCHAR = 20;

   @Override
   public void onCreate(Bundle savedInstanceState) {
```

```
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      msg = (EditText) findViewById(R.id.msg);
      msg.addTextChangedListener(this);
      textIndicator = (TextView) findViewById(R.id.indicator);
      send = (Button) findViewById(R.id.send);
   }

   @Override
   public void afterTextChanged(Editable s) {
      int numChar = msg.getText().toString().length();
      int leftChar = NUMMAXCHAR - numChar;
      if (leftChar >= 0) {
         textIndicator.setText(Integer.toString(leftChar) + "
caracteres restantes");
         textIndicator.setTextColor(Color.GREEN);
         send.setEnabled(true);
      } else {
         textIndicator.setTextColor(Color.RED);
         textIndicator.setText(Integer.toString(Math.abs(leftChar))
+ " en exceso");
         send.setEnabled(false);
      }

   }

   @Override
   public void beforeTextChanged(CharSequence s, int start, int count,
         int after) {
   }

   @Override
   public void onTextChanged(CharSequence s, int start, int
before, int count) {
   }
}
```

Observaciones:

- La clase implementa la interfaz **TextWatcher**.
- El método **addTextChangedListener** permite asociar un **TextWatcher** al campo de texto de entrada.
- Todos los componentes de la interfaz se obtienen en el método **onCreate**.
- En el método **afterTextChanged** se realizan las operaciones siguientes:
  - Se calcula el número de caracteres restantes (variable leftChar).
  - Si el número de caracteres restantes es positivo, se especifica el mensaje que se mostrará en la etiqueta y se activa el botón de envío.
  - En caso contrario, se especifica otro mensaje y se desactiva el botón de envío.

Con lo que se obtendrá:


19:05
TextWatcherEjemplo
un ejemplo
10 caracteres restantes
Enviar


19:05
TextWatcherEjemplo
un ejemplo de TextWatcher
5 caracteres en exceso
Enviar

Notificationes

La barra de notificaciones permite tener al usuario informado sobre los eventos considerados importantes por las aplicaciones instaladas en su teléfono, tales como un nuevo mensaje, una llamada perdida...

Las notificaciones son un elemento muy importante de cualquier aplicación Android. Permiten advertir al usuario algún evento importante. Sobre todo, no hay que abusar del uso de notificaciones, ya que podría molestar al usuario y éste vería obligado a desinstalar la aplicación.

A continuación se muestra una lista exhaustiva de los casos en que se recomienda usar una notificación:

- Un evento asociado al tiempo y en interacción con otra persona (mensaje, email, agenda, mensajería instantánea, etc.).
- Un evento informando al usuario de una alerta suscrita en su aplicación (por ejemplo, si está suscrito a los resultados de un equipo o a un flujo de información).
- Para informar al usuario de la ejecución de un servicio con el que puede interactuar (reproducción de música, etc.).

No debe usar la barra de notificaciones para:

- Mostrar un mensaje publicitario.
- Avisar de la disponibilidad de una actualización de su aplicación (esto es responsabilidad del market).
- Indicar al usuario información que no le concierne directamente.
- Advertir al usuario acerca de un error en su aplicación.
- Advertir al usuario sobre la ejecución de un servicio con el que no puede interactuar.

Cuando el usuario hace clic en una notificación, una actividad de la aplicación en cuestión debe abrirse en un sitio que permita al usuario ver el evento causante de la notificación e interactuar con él. Por ejemplo, si la notificación indica la recepción de un mensaje, debería redirigir al usuario a una pantalla que permita leer y responder dicho mensaje.

No oculte la barra de notificación mostrando su aplicación a pantalla completa, excepto si es realmente necesario (reproducción de vídeos, juegos, etc.).

1. Apariencia

Una notificación puede estar compuesta por los siguientes elementos:

- Una **imagen principal**: imagen que representa la notificación. Por ejemplo, la imagen de un contacto en la recepción del mensaje.
- El **título de la notificación**: por ejemplo, el nombre de la persona que ha enviado el mensaje.
- El **mensaje**: se corresponde con la descripción de la notificación.
- La **hora** (**opcional**): se corresponde con el momento en que se produjo el evento que ha creado la notificación.
- Una **imagen secundaria** (**opcional**): puede representar el icono de su aplicación o el número de notificaciones recibidas.

Aquí se muestra un ejemplo de notificación de la aplicación **Gtalk**:

2. Implementación

a. Crear una notificación

Este ejemplo muestra una vista que tiene un botón que permite crear una notificación cuando el usuario hace un clic.

- Agregue un botón a la interfaz principal de la aplicación:

El

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/addNotification"
        android:id="@+id/addNotification" />

</LinearLayout>
```

método **onCreate** es sencillo, obtiene el botón declarado en la interfaz y le asigna un listener para la gestión del clic. Cada clic se corresponderá con una llamada al método **createNotification**.

```
private int notificationId = 1;

  public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   Button addNotificationBtn = (Button)
findViewById(R.id.addNotification);
   addNotificationBtn.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
         createSimpleNotification();
      }
   });
}
```

- A continuación, implemente el método **createNotification** tal y como se muestra a continuación:

Para

```
protected void createSimpleNotification() {
      final NotificationManager notificationManager =
(NotificationManager) getSystemService(NOTIFICATION_SERVICE);

      final Intent notificationIntent = new
Intent(this,Ch7_SimpleNotificationExempleActivity.class);

       final PendingIntent notificationPendingIntent =
PendingIntent.getActivity(this,
         REQUEST_CODE, notificationIntent,
         PendingIntent.FLAG_ONE_SHOT);

      Notification.Builder notificationBuilder = new
Notification.Builder(this)
      .setWhen(System.currentTimeMillis())
      .setTicker(getResources().getString(
            R.string.notification_launching_title))
      .setSmallIcon(R.drawable.ic_launcher)
      .setContentTitle(
```

```
getResources().getString(R.string.notification_title))
   .setContentText(getResources().getString(R.string. notification_desc))
   .setContentIntent(notificationPendingIntent);

      notificationManager.notify(notificationId,
notificationBuilder.getNotification());
}
```

facilitar la creación de notificaciones, Android dispone de una clase llamada**NotificationManager**. Esta clase permite obtener una instancia del gestor de notificaciones. Para ello, hay que utilizar el método **getSystemService** que permite obtener los distintos servicios y gestores ofrecidos por Android pasándole como parámetro el nombre del servicio deseado (en este caso se usa el servicio **NOTIFICATION_SERVICE**).

El segundo paso consiste en crear un **PendingIntent** (véase el capítulo Principios de programación - Componentes Android) que permite especificar qué vista se desencadenará cuando el usuario haga clic en la notificación.

El método **getActivity** permite crear un pendingIntent correspondiente a una actividad. Este método requiere 4 parámetros:

- El contexto actual.
- Un código identificador de su petición.
- El intent que representa la actividad que se iniciará.
- Una opción entre:
  - **FLAG_CANCEL_CURRENT**: si el intent ya existe, el anterior se destruirá por la creación del nuevo.
  - **FLAG_NO_CREATE**: si el intent no existe, no se creará y el método devolverá null.
  - **FLAG_ONE_SHOT**: el intent sólo podrá usarse una vez.
  - **FLAG_UPDATE_CURRENT**: si el intent ya existe, se conservará su instancia y ésta se actualizará.

Ahora, hay que crear una notificación con un **builder**. Para ello, cree una nueva instancia del**builder**. A continuación, puede encadenar las llamadas a los métodos deseados. En el ejemplo, los métodos utilizados son:

- **setSmallIcon**: permite especificar el icono de la notificación.
- **setTicker**: permite especificar el texto que se mostrará en el momento en que se produzca la notificación.
- **setWhen**: permite especificar el tiempo que se mostrará la notificación.**System.currentTimeMillis** significa que la hora mostrada será la correspondiente al desencadenamiento de la notificación.
- **setContentTitle**: permite especificar el título de la notificación.
- **setContentText**: permite especificar la descripción de la notificación.

El último paso consiste en desencadenar la notificación. Para ello, utilice el método **notify** de la clase **NotificationManager**. Este método recibe como parámetro un valor entero que representa el identificador de la notificación y la notificación obtenida usando el método **getNotification** del builder.

Con lo que se obtendrá:

b. Personalizar una notificación

Puede personalizar la apariencia de una notificación mediante un layout específico.

- Para ello, cree un archivo XML que se corresponda con la interfaz de la notificación. En el siguiente ejemplo, la notificación se compone de un texto y dos imágenes.

El

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:weightSum="100" >

    <ImageView
        android:id="@+id/img"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="5dp"
        android:layout_weight="20" />

    <TextView
        android:id="@+id/txt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="50"
        android:padding="15dp"/>

    <ImageView
        android:id="@+id/img2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginTop="15dp"
        android:layout_weight="30"
        />

</LinearLayout>
```

siguiente código permite cargar la interfaz para poderla usar como vista de personalización de la notificación.

- Para comenzar, cree una instancia de la clase **RemoteViews**. Esta clase se inicializa mediante su constructor, que recibe como parámetro el nombre del package en el que se encuentra el layout y una referencia al layout que se cargará.

La

```
package com.eni.android.notification;

import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
```

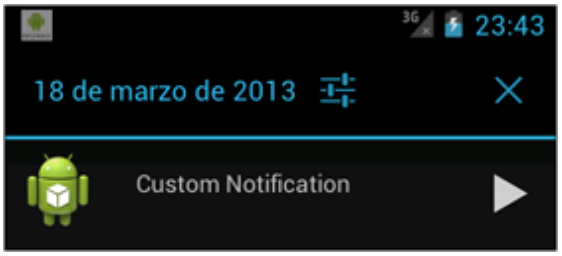
```
import android.widget.RemoteViews;

public class Cap7_CustomNotificationEjemploActivity extends Activity {
    private static int NOTIFICATION_ID = 0;
    private final static int REQUEST_CODE = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button addNotification = (Button) findViewById
(R.id.addNotification);
        addNotification.setOnClickListener(new OnClickListener() {
              @Override
              public void onClick(View v) {
                    createNotification();
              }
        });
    }

    protected void createNotification() {
        final NotificationManager mNotification =
(NotificationManager) getSystemService(NOTIFICATION_SERVICE);
        RemoteViews contentView = new RemoteViews(getPackageName(),
                 R.layout.custom_notification);
        contentView.setImageViewResource(R.id.img,
R.drawable.ic_launcher);
        contentView.setTextViewText(R.id.txt,
getResources().getString(R.string.custom_notification));
        contentView.setImageViewResource(R.id.img2, R.drawable.play);

        final PendingIntent pendingIntent =
PendingIntent.getActivity(this, REQUEST_CODE, new Intent(this,
                      Cap7_CustomNotificationEjemploActivity.class),
                      PendingIntent.FLAG_ONE_SHOT);

        Notification.Builder builder = new Notification.Builder(this)
                        .setSmallIcon(R.drawable.notification_icon)
                        .setTicker(
                               getResources().getString(
    R.string.notification_launching_title))
                        .setWhen(System.currentTimeMillis())
                        .setContentTitle(
getResources().getString(R.string.notification_title))
                        .setContentText(
getResources().getString(R.string.notification_desc))
                        .setContentIntent(pendingIntent)
.setContent(contentView);

          mNotification.notify(NOTIFICATION_ID,
builder.getNotification());
    }
}
```

clase **RemoteViews** se corresponde con una vista que puede cargarse desde otra aplicación. Éste es el motivo por el cual se especifica el nombre del package en el que se encuentra la vista deseada.

Para asociar la vista personalizada a la notificación, utilice el método **setContent** pasándole como parámetro el **RemoteViews** declarado anteriormente.

```
builder.setContent(notificationCustomView) ;
```

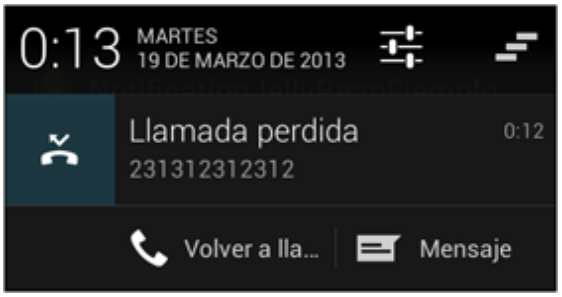
Con lo que se obtendrá:



3. Notificaciones en Jelly Bean

Las nuevas versiones de Android (4.1 y 4.2 - Jelly Bean) aportan muchas novedades en lo referente al sistema de notificaciones. Ahora, el usuario podrá:

- Saber qué aplicación es el origen de una notificación.
- Impedir que una aplicación genere notificaciones.
- Interactuar con una notificación directamente desde la barra de notificaciones (responder a un SMS, recuperar un contacto).
- Expandir o reducir una notificación para ver más o menos detalles de ésta.
- Las notificaciones se ordenarán por prioridad y, a igual prioridad, por fecha de creación.



Una notificación ahora puede tener las siguientes propiedades:

- **Prioridad**: cada notificación tiene una prioridad cuyo valor está entre los siguientes valores:**PRIORITY_MAX**, **PRIORITY_HIGH**, **PRIORITY_DEFAULT**, **PRIORITY_LOW**,**PRIORITY_MIN**).
  - Por defecto, una notificación tiene la prioridad **PRIORITY_DEFAULT**.
  - La prioridad **PRIORITY_MIN** tiene un comportamiento particular, la notificación se presenta en la barra de notificaciones, pero cuando se desencadena, el usuario no será avisado (no aparecerá en la barra de estado).
  - No abuse de las prioridades **PRIORITY_MAX** o **PRIORITY_HIGH**.
- **Nuevas zonas de contenidos**: permite agregar dos nuevos tipos de contenido en una notificación.
  - Puede agregar una gran etiqueta de texto para mostrar, por ejemplo, el contenido de un email o de un mensaje.
  - Puede agregar una gran imagen en una notificación.
- **Actions**: permite agregar acciones para permitir al usuario interactuar directamente con una notificación. Por ejemplo, en el caso de una llamada perdida, ofrecer acciones que permitan buscar al contacto o enviarle un mensaje.

A continuación se muestra un ejemplo de notificación creada cuando el usuario hace clic en un botón de una interfaz:

Este

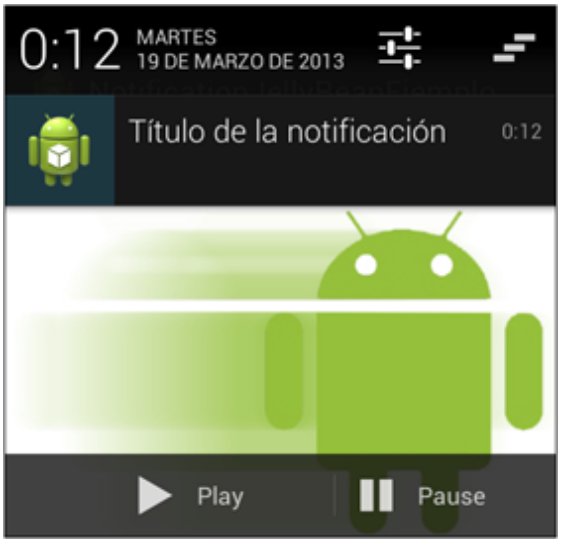
```
protected void createNotification() {
          final NotificationManager mNotification =
(NotificationManager) getSystemService(NOTIFICATION_SERVICE);
```

```
          final Intent launchNotifiactionIntent = new
Intent(this,
                        MainActivity.class);
          final PendingIntent pendingIntent =
PendingIntent.getActivity(this,
                        REQUEST_CODE, launchNotifiactionIntent,
                        PendingIntent.FLAG_ONE_SHOT);

          Notification.Builder builder = new
Notification.Builder(this)
                        .setWhen(System.currentTimeMillis())
                        .setTicker(
                                    getResources().getString(

    R.string.notification_launching_title))
                        .setSmallIcon(R.drawable.ic_launcher)
                        .setContentTitle(

    getResources().getString(R.string.notification_title))
                        .setContentText(

    getResources().getString(R.string.notification_desc))
                        .setContentIntent(pendingIntent)
                        .addAction(
                                R.drawable.play,
                                "Play",

    PendingIntent.getActivity(getApplicationContext(), 0,
                                          getIntent(), 0, null))
                        .addAction(
                                R.drawable.pause,
                                "Pause",

    PendingIntent.getActivity(getApplicationContext(), 0,
                                                    getIntent(),
0, null));

          Notification notification = new
Notification.BigPictureStyle(builder)
                         .bigPicture(
    BitmapFactory.decodeResource(getResources(),

    R.drawable.android_logo)).build();

           mNotification.notify(NOTIFICATION_ID, notification);
    }
```

ejemplo de creación de notificación es muy similar a los ejemplos anteriores. La única diferencia es el uso de los métodos:

- **setAction**: especifica las acciones accesibles desde la notificación.
- **BigPictureStyle**: define una gran imagen para mostrar en la notificación.

Con lo que se obtendrá:


0:12
MARTES
19 DE MARZO DE 2013
Título de la notificación
0:12
Play
Pause

Gestión de la rotación

Una de las características específicas del desarrollo en Android es la gestión de la rotación en una aplicación (modos vertical y apaisado).

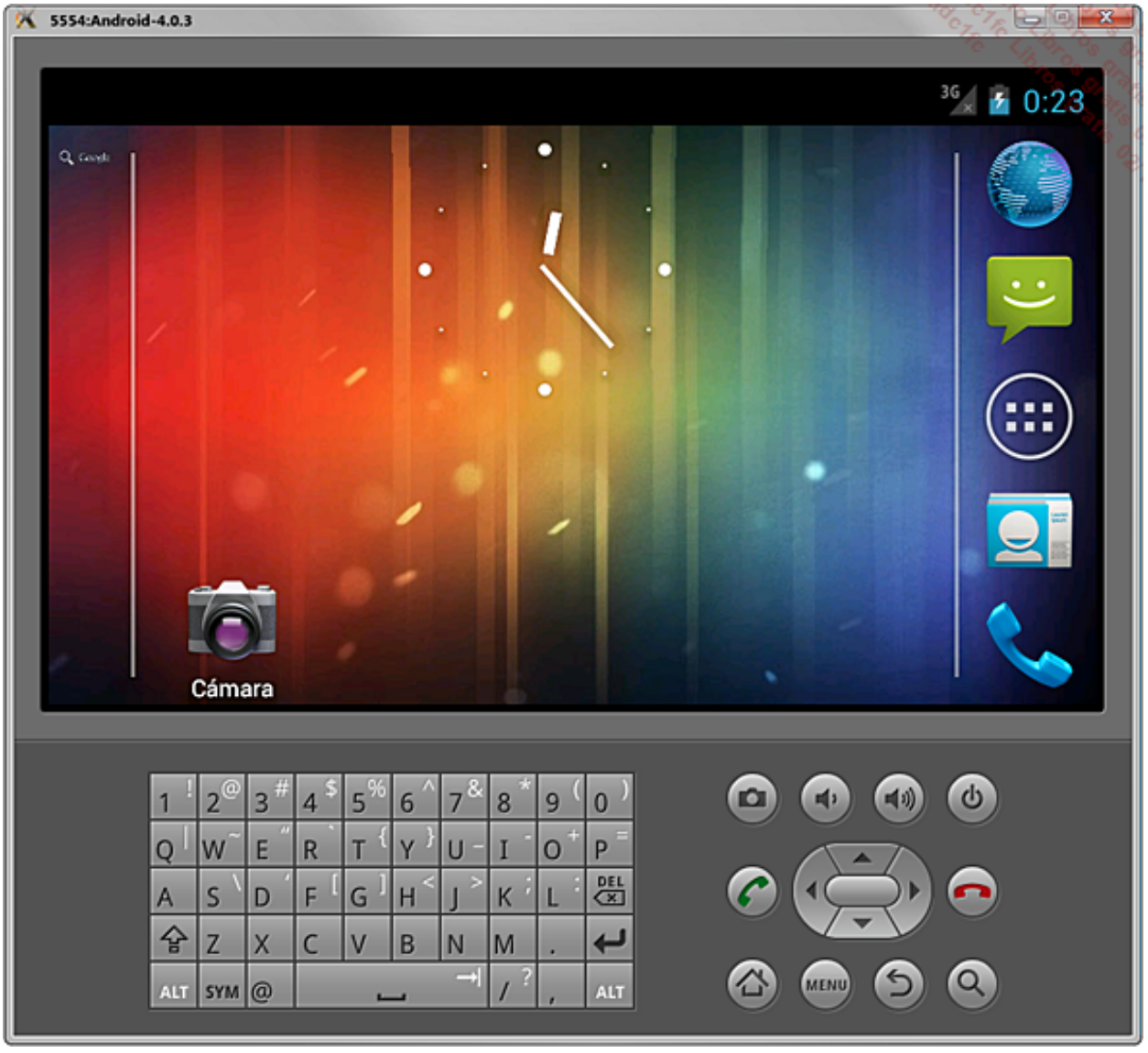
En la rotación de un teléfono de modo vertical a modo apaisado, Android destruye la vista y vuelve a crear la vista correspondiente a la aplicación en el nuevo modo. Debe ofrecer una visualización alternativa de su aplicación.

1. Gestión del modo apaisado

La carpeta **layout**, presente en la carpeta **res**, contiene las interfaces en modo vertical. Para gestionar el modo apaisado, debe crear una carpeta llamada **layout-land** que contendrá la interfaz en modo apaisado.

Debe volver a diseñar sus interfaces en modo apaisado para adaptar los componentes, su posición, su tamaño y el tamaño de la fuente a la versión apaisada de su aplicación.

Para alternar entre el modo vertical y el modo apaisado en el emulador, hay que pulsar [Ctrl] [F12] o la tecla [9] del keypad.



a. Ejemplo

En el siguiente ejemplo, se va a crear una interfaz de conexión en modo vertical y en modo apaisado.

La interfaz se compone de:

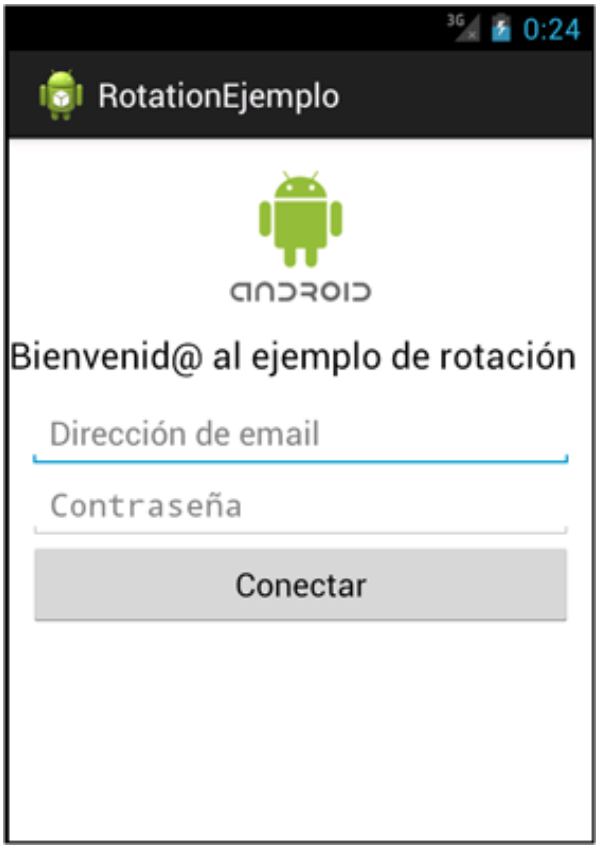
- Una imagen (opcional).
- Un texto.
- Dos campos de edición.
- Un botón.

Comience por el modo vertical, con el que obtendrá:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@color/white"
    android:orientation="vertical" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="@dimen/margin"
        android:src="@drawable/android" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin"
        android:gravity="center_horizontal"
        android:text="@string/hello"
        android:textColor="@color/black"
        android:textSize="20sp" />

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/margin"
        android:layout_marginRight="@dimen/margin"
        android:layout_marginTop="@dimen/margin"
        android:hint="@string/email"
        android:inputType="textEmailAddress"
        android:textColor="@color/black" />

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/margin"
        android:layout_marginRight="@dimen/margin"
        android:hint="@string/pass"
        android:inputType="textPassword"
        android:textColor="@color/black" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/margin"
        android:layout_marginRight="@dimen/margin"
```

```
        android:text="@string/connect"
        android:textColor="@color/black"/>

</LinearLayout>
```

Obtendrá el siguiente resultado:



Y por lo que respecta al modo apaisado:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@color/white"
    android:orientation="vertical" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin"
        android:layout_gravity="center_horizontal"
        android:text="@string/hello"
        android:textColor="@color/black"
        android:textSize="20sp" />

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/margin"
        android:layout_marginRight="@dimen/margin"
        android:layout_marginTop="@dimen/margin"
        android:hint="@string/email"
        android:inputType="textEmailAddress"
```

```
        android:textColor="@color/black" />

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/margin"
        android:layout_marginRight="@dimen/margin"
        android:hint="@string/pass"
        android:inputType="textPassword"
        android:textColor="@color/black" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/margin"
        android:layout_marginRight="@dimen/margin"
        android:text="@string/connect"
        android:textColor="@color/black"/>

</LinearLayout>
```

Hay que reorganizar la interfaz para preservar la experiencia del usuario. En el ejemplo, la imagen se ha eliminado para ganar espacio.

También puede reducir el tamaño de los espaciados, el tamaño de los textos, etc.

2. Bloquear la vista en un solo modo

Si fuera necesario, puede bloquear vistas en el modo vertical o en el modo apaisado. Este método de gestión de la rotación no se recomienda y no debe utilizarse salvo que sea necesario (videojuegos, etc.), ya que reduce significativamente la experiencia de usuario de la aplicación.

Cuando bloquea una vista en un modo, se produce un problema por la compatibilidad entre smartphones y tablets ya que el modo de uso por defecto de un teléfono es el modo vertical mientras que el de una tablet es el modo apaisado.

Para realizar esta operación, debe añadir el atributo **android:screenOrientation** en su manifiesto, en la línea correspondiente a la declaración de la actividad afectada. Este atributo puede adquirir los siguientes valores:

- **unspecified** (valor por defecto): deja al sistema Android elegir la orientación que se mostrará.

- **user**: la orientación preferida del usuario.
- **behind**: la misma orientación que la de la actividad anterior.
- **landscape**: bloquea en modo apaisado.
- **portrait**: bloquea en modo vertical.
- **reverseLandscape**: bloquea en modo apaisado, sin embargo este modo se invierte con el modo apaisado natural (el modo invertido se corresponde con una rotación de 90º del dispositivo desde su modo natural).
- **reversePortrait**: bloquea en modo vertical, sin embargo este modo se invierte con el modo vertical natural.
- **sensorLandscape**: modo apasiado, pero puede tener el valor apaisado normal o invertido. El valor se calcula mediante el sensor del teléfono.
- **sensorPortrait**: modo vertical, pero puede tener el valor vertical normal o invertido. El valor se calcula mediante el sensor del teléfono.
- **sensor**: la orientación se define dinámicamente mediante el sensor del teléfono (2 orientaciones posibles).
- **fullsensor**: la orientación se define dinámicamente mediante el sensor del teléfono (4 orientaciones posibles).
- **nosensor**: la orientación se define sin tener en cuenta el sensor de orientación.

```
<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name" >
    <activity
        android:name=". EjemploActivity"
        android:label="@string/app_name"
        android:screenOrientation="portrait">
    </activity>
</application>
```

3. Gestionar manualmente la rotación de pantalla

En algunos casos, necesitará sobrecargar el comportamiento de Android en la rotación de la aplicación y, para ello, deberá especificar un comportamiento personalizado.

Para realizar esta acción, hay que añadir a la actividad deseada el atributo**android:configChanges="orientation"** (en el archivo de manifiesto).

En la actividad deseada, puede sobrecargar el método **onConfigurationChanged()** para especificar el comportamiento de su actividad tras la rotación.

Puede sobrecargar, de igual modo, otras propiedades del dispositivo.

A partir del ejemplo anterior, agregue el atributo **configChanges** en el archivo de manifiesto. Obtendrá:

```
<application
     android:icon="@drawable/ic_launcher"
     android:label="@string/app_name" >
     <activity
         android:name=".Cap7_RotacionManualEjemploActivity"
         android:label="@string/app_name"
         android:configChanges="orientation">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
```

```
            <category
android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
     </activity>
</application>
```

Inicie la aplicación y haga clic en el primer campo para introducir texto. Una vez haya terminado de introducir el texto, realice una rotación de su emulador. Podrá observar que el texto introducido desaparece. Esto es debido a la sobrecarga de la rotación, que no permite a Android realizar la copia de seguridad del texto introducido.


Recordatorio: la rotación destruye totalmente su actividad y la recrea en el modo adecuado.

Para poder guardar el estado de sus campos, debe utilizar los siguientes dos métodos:

- **onSaveInstanceState**: permite guardar el estado de la actividad.
- **onRestoreInstanceState**: permite restaurar el estado de la actividad.

En el ejemplo:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    EditText email = (EditText) findViewById(R.id.email);
    String emailTxt = email.getText().toString();
    outState.putString("EMAIL", emailTxt);
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
    super.onRestoreInstanceState(savedInstanceState);
    EditText email = (EditText) findViewById(R.id.email);
    email.setText(savedInstanceState.getString("EMAIL"));
}
```

- En el método **onSaveInstanceState**, hay que guardar el estado del campo de texto en el **Bundle** que se pasa por parámetro.
- En el método **onRestoreInstanceState**, hay que restaurar el estado del campo de texto a partir del **Bundle** que se pasa por parámetro.

Ahora, puede comprobar de nuevo el ejemplo y observar que el texto se conserva en las rotaciones.

Creación de vistas personalizadas

La creación de la interfaz Android no se limita a componentes ya existentes. Se pueden crear nuevos componentes, nuevos layouts y nuevas vistas totalmente personalizadas.

Puede:

- Sobrecargar un componente ya existente para modificarlo, mejorarlo o añadirle funcionalidades. Para ello, su componente personalizado debe heredar de la clase que representa al componente deseado y sobrecargar los siguientes métodos: Constructor,**onDraw** y Gestión de eventos. El método onDraw permite diseñar el componente.
- Combinar componentes existentes para crear nuevos componentes. Puede heredar de la clase**View** y crear una vista reutilizable con los componentes deseados.
- Crear nuevos componentes desde cero.

Tomemos el ejemplo de una vista compuesta de un texto vertical y de una imagen. Para comenzar, hay que crear una clase que extienda de la clase **View**.

```
public class MyCustomView extends View {
```

El segundo paso consiste en sobrecargar los dos constructores para inicializar todas las variables necesarias en el diseño de la nueva vista.

```
public MyCustomView(Context context) {
   super(context);
   init();
}
public MyCustomView(Context context, AttributeSet attrs) { super(context, attrs);
   init();
}
```

El método **init()** permite inicializar los distintos elementos necesarios en el resto del tratamiento (colores, textos, recursos...).

```
private void init() {
   Resources res = getResources();

   textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   textPaint.setColor(res.getColor(R.color.button_text));
   textPaint.setTextSize(25);

   bitmap = BitmapFactory.decodeResource(getResources(),
               R.drawable.ic_launcher);

}
```

Los dos elementos necesarios para continuar el tratamiento son el estilo, que servirá para escribir el texto deseado (color y tamaño del texto), así como la imagen que se mostrará.

La última etapa consiste en sobrecargar el método **onDraw**, que permite dibujar los elementos deseados.

```
@Override
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);
   canvas.drawBitmap(bitmap, 130, 10, null);
   canvas.rotate(90);
   canvas.drawText(getResources().getString(R.string.verticalText),
```

```
0, 0, textPaint);
   canvas.save();
}
```

El diseño se realiza siguiendo los pasos descritos:

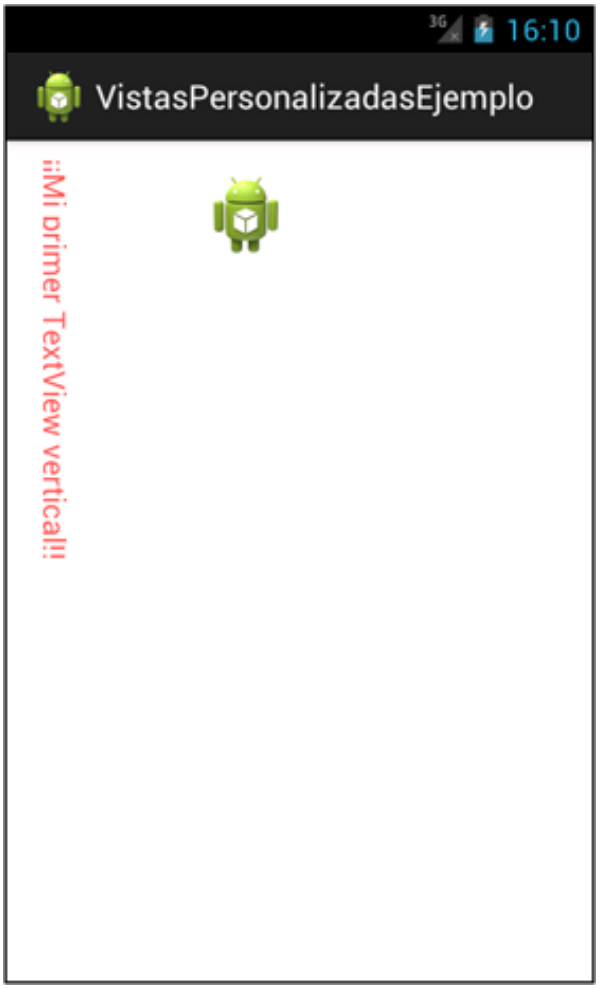
- Pintado de la imagen mediante el método **drawBitmap**, que recibe como parámetro el bitmap que se mostrará, la posición X de la imagen, la posición Y de la imagen así como el color para pintarla. Puede pasar null para indicar que no desea ningún color específico.
- Rotación de 90° de la zona de diseño. Sólo los diseños que se coloquen después de la rotación se verán afectados por ésta última.
- Dibujo del texto mediante el método **drawText**, que recibe como parámetro el texto que se desea mostrar, las posiciones X e Y del texto así como el estilo del dibujo (color, texto...).
- Para finalizar, guarde el estado del canvas para guardar el diseño realizado.

Para utilizar su vista personalizada en un archivo de vista se realiza tal y como se muestra a continuación:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <com.eni.android.vues.personnalisees.MyCustomView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:layout_marginLeft="20dp" />

</LinearLayout>
```

Observe que se utiliza la vista personalizada como un componente nativo más. Basta, simplemente, con especificarle la ruta al componente.

Este ejemplo producirá el siguiente resultado:


16:10
VistasPersonalizadasEjemplo
¡¡Mi primer TextView vertical!!

Listas

En Android, una lista representa un conjunto de elementos que se muestran los unos a continuación de los otros.

Cada elemento de una lista puede tener de una a tres filas y cada fila puede personalizarse con diferentes componentes (**TextView**, **Button**, **ImageView**...).

Para crear una lista dispone de dos métodos. La actividad que contiene la lista puede:

- O bien heredar de la clase **ListActivity**.
- O bien heredar de la clase **Activity**.

Para insertar datos en una lista se utiliza un **adapter** (adaptador). Éste permite asociar datos a una vista que extienda de la clase **AdapterView**, así es fácil acceder a los datos almacenados (leer, agregar, eliminar, modificar...) en una vista.

Android ofrece dos tipos de adapters:

- **ArrayAdapter**: permite rellenar una lista a partir de una tabla o de una colección.
- **SimpleCursorAdapter**: permite rellenar una lista a partir de una base de datos.

También puede crear su propio adapter simplemente heredando de la clase **BaseAdapter** o de un adaptador ya existente.

1. Creación de una lista

a. ListActivity

El primer método consiste en crear una lista que herede de la clase **ListActivity**.

Para comenzar, cree un archivo XML que represente una vista que contenga solamente una lista (carpeta **layout**).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ListView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@android:id/list" />

</LinearLayout>
```

Para poder operar una **ListView** mediante una **ListActivity**, su lista debe tener obligatoriamente como identificador **@android:id/list**.

A continuación, declare una actividad que herede de la clase **ListActivity**.

```
public class ListActivityExampleActivity extends ListActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

```
    }
}
```

Por el momento, la lista está vacía y no contiene ningún dato. Vamos a utilizar una tabla de cadenas de caracteres como origen de datos.

```
private String[] androidVersion = {"Cupcake", "Donut", "Eclair",
"Froyo", "Gingerbread", "Honeycomb", "Ice Cream Sandwich",
"Jelly Bean"};
```

- Cree un **ArrayAdapter** para inyectar en la lista los datos provenientes de la tabla declarada anteriormente.

```
ArrayAdapter adapter = new ArrayAdapter(this,
                android.R.layout.simple_list_item_1,
androidVersion);
```

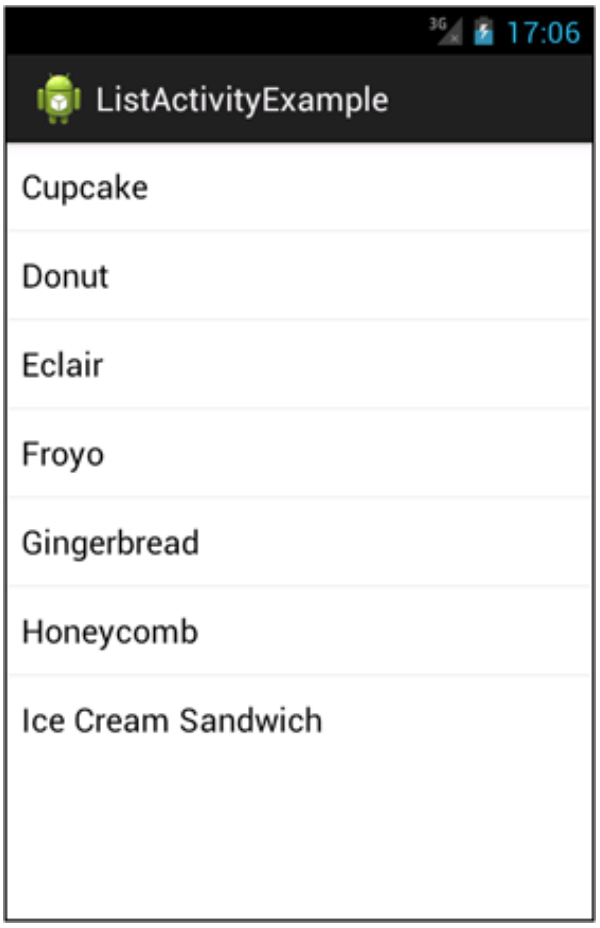
El método que permite construir un **ArrayAdapter** recibe tres parámetros:

- El primer parámetro representa el contexto de la actividad actual.
- El segundo parámetro representa el layout que se aplicará a cada fila de la lista. Puede o bien crear un layout personalizado, o bien utilizar alguno de los proporcionados por Android.
- El tercer parámetro representa la tabla de datos que se insertará en la lista.

- El último paso consiste en asociar el adaptador, que contiene los datos que se inyectarán, a la lista. Para ello, se utiliza el método **setListAdapter**:

```
SetListAdapter(adapter);
```

A continuación, puede probar este ejemplo. Debería obtener el siguiente resultado:



b. ListView

La segunda forma de crear una lista es, simplemente, usar una actividad.

- Para ello, hay que crear el archivo XML que representa la lista.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ListView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/myList" />

</LinearLayout>
```

Este método ofrece la posibilidad de especificar un identificador a su elección.

- Modifique la actividad para recuperar la instancia de la lista (mediante el método**findViewById**).

```
ListView myList = (ListView) findViewById(R.id.myList);
```

Como en el caso anterior, el ejemplo utiliza una tabla de cadenas de caracteres así como un**ArrayAdapter**.

- Para finalizar, hay que asociar el adaptador a la lista.

```
myList.setAdapter(adapter);
```

El uso de una ListActivity es aconsejable si su vista sólo contiene una lista.

c. Adapter y lista personalizada

También puede crear adapters personalizados para gestionar mejor la visualización y los datos de una lista.

A continuación se muestra un ejemplo que permite mostrar una lista que contiene el nombre y el número de las diferentes versiones de Android usando un adapter personalizado.

- Para comenzar, cree la clase **AndroidVersion**, que representa cualquier versión de Android.

```
public class AndroidVersion {

    private String versionName;
    private String versionNumber;

    public String getVersionName() {
       return versionName;
    }
    public void setVersionName(String versionName) {
       this.versionName = versionName;
    }
    public String getVersionNumber() {
       return versionNumber;
```

```
    }
    public void setVersionNumber(String versionNumber) {
       this.versionNumber = versionNumber;
    }


}
```

El ejemplo utiliza un adapter personalizado que hereda de la clase **ArrayAdapter** (ya que utiliza una tabla para inyectar datos en la lista).

- Para ello, cree un archivo que representará la lista.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ListView
        android:id="@+id/myList"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

Para personalizar la lista y hacerla más rica y agradable, hay que crear un layout personalizado que servirá para especificar la interfaz correspondiente a cada fila de la lista.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:padding="6dip" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="6dp"
        android:src="@drawable/list_icon" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:orientation="vertical" >

        <TextView
            android:id="@+id/title"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:gravity="center_vertical"
            />

        <TextView
            android:id="@+id/description"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            />

    </LinearLayout>

</LinearLayout>
```

Por lo tanto, cada fila de la lista se compone de los siguientes elementos:

- Un **LinearLayout** horizontal.
- Una imagen.
  - La imagen se ubica en la carpeta **drawable**.
  - La imagen tiene un espaciado exterior (**margin**) derecho de **6dp**.
- Se utiliza un segundo **LinearLayout** (vertical) para mostrar los dos textos (título y descripción).
- Un primer texto que indica el título de la fila.
- Un segundo texto que indica la descripción de la fila.

A continuación, cree una clase que represente el adapter personalizado. Tendrá las siguientes características específicas:

- Heredará de la clase **ArrayAdapter**, ya que los datos se completarán usando una tabla.
- Cada elemento de la lista representará una versión de Android.
- Tendrá los dos siguientes métodos:
  - Un constructor.
  - Un método **getView**: cada llamada a este método permitirá obtener una fila (dato y valor) de la lista que se encontrará en una posición determinada.

```
public class AndroidAdapter extends ArrayAdapter<AndroidVersion>
{

    ArrayList<AndroidVersion> androidVer;
    int viewRes;

    public AndroidAdapter(Context context, int
textViewResourceId,
          ArrayList<AndroidVersion> versions) {
      super(context, textViewResourceId, versions);
      this.androidVer = versions;
      this.context = context;
      this.viewRes = textViewResourceId;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup
parent) {
      View v = convertView;
      if (v == null) {
         LayoutInflater vi = (LayoutInflater)
context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
         v = vi.inflate(viewRes, parent, false);
      }
      AndroidVersion o = androidVer.get(position);
      if (o != null) {
         TextView tt = (TextView) v.findViewById(R.id.title);
         TextView bt = (TextView)
v.findViewById(R.id.description);
         if (tt != null) {
             tt.setText("Nombre de la versión: " +
o.getVersionName());
         }
         if (bt != null) {
             bt.setText("Número de la versión: " +
```

```
o.getVersionNumber());
         }
      }
      return v;
   }
}
```

El constructor sirve para almacenar los tres parámetros necesarios para implementar el adapter:

- El contexto de la vista.
- El layout correspondiente a la vista personalizada, aplicado a cada fila de la lista.
- La tabla que representa los datos que se insertarán en la lista.

El método **getView(int position, Viwe convertView, ViewGroup parent)**:

- Este método permite obtener la vista a partir de una fila determinada (parámetro **position**).
- La vista que representa la lista se pasa por parámetro al método (parámetro **convertView**).
- La vista padre se pasa por parámetro (parámetro **parent**).
- La vista personalizada debe cargarse mediante el método **inflate**. Una vez se ha cargado, se pasará por parámetro al método **getView** (parámetro **convertView**) en las siguientes llamadas, lo que reduce el número de llamadas al método **inflate** (llamada con bastante coste).
- A continuación, hay que obtener el texto correspondiente a la fila que se desea mostrar.
- Por último, hay que obtener los dos **TextView** (título y descripción de una fila) para rellenarlos con los datos obtenidos en el paso anterior.

Puede ir más allá en la optimización de una lista mediante el sistema de holder.

Para acabar, cree una actividad que permita inicializar la vista, el adapter personalizado y los datos.

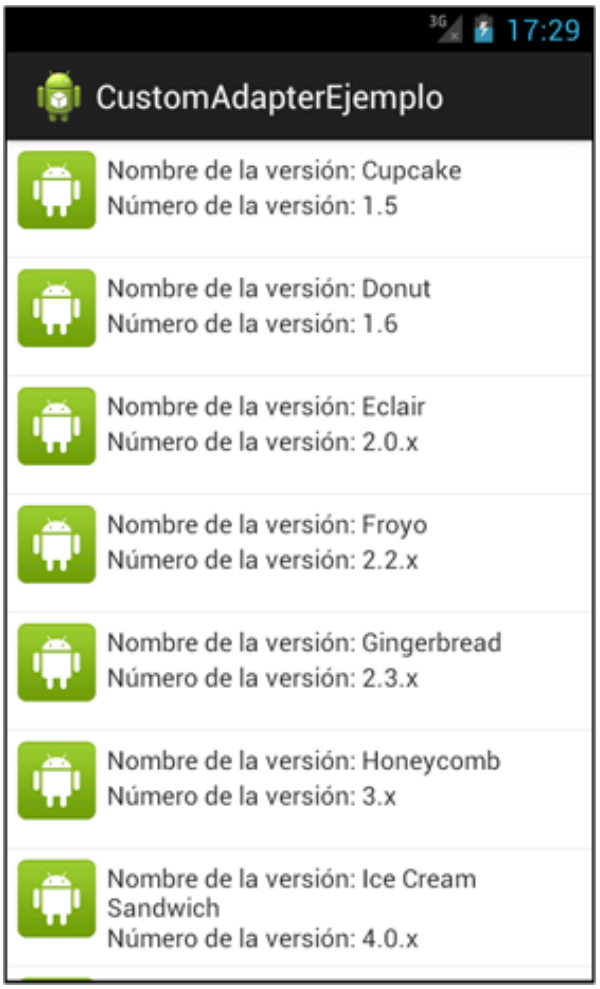
```
public class CustomAdapterExampleActivity extends Activity {

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      ArrayList<AndroidVersion> androidList = new
ArrayList<AndroidVersion>();

         initList(androidList);

         AndroidAdapter adapter = new AndroidAdapter(this,
R.layout.list_layout, androidList);
         final ListView list = (ListView)
findViewById(R.id.myList);
         list.setAdapter(adapter);
    }

   private void initList(ArrayList<AndroidVersion> androidList)
{
      AndroidVersion version = new AndroidVersion();
      version.setVersionName("Cupcake");
      version.setVersionNumber("1.5");
      androidList.add(version);

      AndroidVersion versionDonut = new AndroidVersion();
      versionDonut.setVersionName("Donut");
      versionDonut.setVersionNumber("1.6");
```

```
        androidList.add(versionDonut);

        AndroidVersion versionEclair = new AndroidVersion();
        versionEclair.setVersionName("Eclair");
        versionEclair.setVersionNumber("2.0.x");
        androidList.add(versionEclair);

        AndroidVersion versionFroyo = new AndroidVersion();
        versionFroyo.setVersionName("Froyo");
        versionFroyo.setVersionNumber("2.2.x");
        androidList.add(versionFroyo);

        .........
    }
}
```

Esta actividad permite:

- Inicializar la lista de datos (método **initList**).
- Inicializar el adapter pasándole los parámetros necesarios para su buen funcionamiento (contexto, vista y datos).
- Obtener la lista y asociarla al adapter.

Con lo que se obtendrá:



Gestión del clic en una lista

Para gestionar el clic en cada uno de los elementos de una lista, utilice el método***setOnItemClickListener*** en la instancia de la lista deseada.

```
list.setOnItemClickListener(new OnItemClickListener() {

   @Override
   public void onItemClick(AdapterView<?> adapter, View v, int
position, long id) {
      AndroidVersion selectedItem = (AndroidVersion)
adapter.getItemAtPosition(position);
      Log.v("CustomAdapterExample", "Elemento seleccionado: "
+ selectedItem.getVersionName());
   }
});
```

El método **onItemClick** recibe cuarto argumentos:

- El adapter sobre el que se ha hecho clic.
- La vista en la que se ha producido el clic.
- La posición del clic en el adapter.
- El identificador de la posición del clic en la vista.

A continuación, obtenga la instancia de la clase **AndroidVersion** que se corresponde con el elemento seleccionado por el usuario para mostrarlo en un Log.

Fragment

Un fragment (véase el capítulo Principios de programación - Componentes Android) es un compoenente que debe adjuntarse a una actividad para poder utilizarse. Su ciclo de vida se parece al de la actividad que lo contiene pero tiene algunas particularidades.

1. Ciclo de vida de un fragment

La primera etapa del ciclo de vida de un fragment se corresponde con el instante en que el fragment se adjunta a la actividad que lo contiene (**onAttach**). A continuación, el fragment se inicializa en la llamada al método **onCreate**, seguido de la creación y de la carga de la interfaz del fragment (método **onCreateView**).

Una vez que la actividad padre y el fragment se han creado, la llamada al método **onActivityCreated**indica el final del ciclo de creación de la interfaz.

El método **onStart** coincide con el paso del fragment a primer plano, seguido de la llamada almétodo **onResume** (mismo funcionamiento que para una actividad - véase el capítulo Principios de programación - Ciclo de vida de una actividad).

Cuando un fragment pasa a estar inactivo, la llamada al método **onPause** permite ejecutar las acciones adecuadas (deshabilitar actualizaciones, listener...). Después de esta llamada, se invoca almétodo **onStop** (el fragmento deja de estar en primer plano).

Si se destruye un fragment, se invocará respectivamente a los siguientes métodos:

- **onDestroyView**: destrucción de la vista.
- **onDestroy**: destrucción del fragment.
- **onDetach**: el fragment se desvincula de la actividad que lo contiene.

2. Ejemplo

En el siguiente ejemplo, se creará una vista compuesta de dos secciones:

- **Aplicación en modo vertical**: compuesta por dos actividades. La primera representa la lista de versiones de Android y la segunda representa la vista de detalles que se muestra cuando el usuario hace clic en un elemento de la lista.
- **Aplicación en modo apaisado**: compuesta por una única actividad pero con dos fragments. En el momento en que el usuario hace clic en la lista que se encuentra en el primer fragment, el segundo automáticamente se actualizará con los detalles.

➔ Para empezar, cree en la carpeta layout un archivo XML que represente la vista en modo apaisado (vista compuesta de dos fragments).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/listFragment"
        android:layout_width="150dip"
        android:layout_height="match_parent"
        class="com.eni.android.fragment.ListFragment" >
</fragment>
```

```
    <fragment
        android:id="@+id/detailFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.eni.android.fragment.DetailFragment" >
    </fragment>

</LinearLayout>
```

Este archivo se compone de dos fragments:

- **Fragment 1**: representa una lista de versiones de Android. Se declara en el archivo**ListFragment** (véase el atributo class).
- **Fragment 2**: representa la vista detallada de una versión de Android. Se declara en el archivo **DetailFragment**.

Ahora, implemente el primer fragment (lista de versiones de Android) heredando de la clase**ListFragment**.

```
public class ListFragment extends android.app.ListFragment {

private String[] values = {"Cupcake", "Donut", "Eclair", "Froyo",
   "Gingerbread", "Honeycomb", "Ice Cream Sandwich", "Jelly Bean"};

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
   }

   @Override
   public void onActivityCreated(Bundle savedInstanceState) {
      super.onActivityCreated(savedInstanceState);
      ArrayAdapter<String> adapter = new
ArrayAdapter<String>(getActivity(),
                android.R.layout.simple_list_item_1, values);
      setListAdapter(adapter);

   @Override
   public void onListItemClick(ListView l, View v, int position,
long id) {
      String item = (String) getListAdapter().getItem(position);
      DetailFragment fragment = (DetailFragment)
getFragmentManager().findFragmentById(R.id.detailFragment);
      if (fragment != null && fragment.isInLayout()) {
          fragment.setText(item);
      } else {
         Intent intent = new
Intent(getActivity().getApplicationContext(),
               DetailActivity.class);
         intent.putExtra("value", item);
         startActivity(intent);
      }

    }
```

El método **OnActivityCreated** se invoca en la creación de un fragment.

- En este método, el adapter se inicializa mediante la tabla **values**.
- Después, la lista se asocia al adapter.

El método **onListItemClick** se sobrecarga para gestionar el clic en un elemento de la lista:

- Se obtiene el elemento que ha recibido el clic usando el adapter.

- Se inicializa el fragment que sirve para mostrar los detalles.
- Si no se ha encontrado ningún problema en la inicialización, significa que la aplicación está en**modo apaisado** y, por lo tanto, que se puede usar el fragment de detalles. En caso contrario la aplicación está **en modo vertical**.
- Si la inicialización funciona correctamente, permite definir el texto que se muestra en el fragment de detalles.
- En caso contrario, hay que ejecutar la actividad de detalles y no el fragment de detalles.

Ahora, hay que implementar el fragment que sirve para mostrar los detalles:

```
public class DetailFragment extends Fragment {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
   }

   @Override
   public void onActivityCreated(Bundle savedInstanceState) {
      super.onActivityCreated(savedInstanceState);
   }

   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup
container,
         Bundle savedInstanceState) {
      View view = inflater.inflate(R.layout.details,
container, false);
      return view;
   }

   public void setText(String item) {
      TextView view = (TextView)
getView().findViewById(R.id.detailsText);
      view.setText(item);
   }
}
```

Una vez que se han realizado estos pasos, el modo apaisado estará correctamente implementado.

El siguiente paso consiste en implementar las interfaces y los tratamientos para el modo vertical. Para ello, hay que crear la carpeta **layout-port** que contendrá una segunda versión del archivo**main.xml**.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/listFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.eni.android.fragment.ListFragment" />

</LinearLayout>
```

En esta versión, el archivo se compone únicamente de un fragment (el que muestra la lista de versiones de Android). Las diferencias entre el archivo **main.xml** en modo vertical y en modo apaisado permiten identificar la orientación en la inicialización de la vista y, por lo tanto, adoptar el comportamiento adecuado en función de cada caso.

- Cree un archivo XML que represente la vista de detalles. Contendrá el fragment que permite mostrar los detalles de una versión de Android en modo vertical.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <fragment
        android:id="@+id/detailFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.eni.android.fragment.DetailFragment" />

</LinearLayout>
```

Finalmente, se han separado los dos fragments que se encontraban en el mismo archivo (modo apaisado) en dos archivos diferentes (modo vertical).

- A continuación, hay que crear la actividad que permite gestionar la vista de detalles.

```
public class DetailActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.details_activity_layout);
      Bundle extras = getIntent().getExtras();
      if (extras != null) {
         String s = extras.getString("value");
         TextView view = (TextView)
findViewById(R.id.detailsText);
         view.setText(s);
      }

   }
}
```

Esta actividad carga el nuevo layout y muestra la cadena de caracteres que se pasa como parámetro (extras).

- Para finalizar, hay que implementar la actividad principal del ejemplo:

```
public class FragmentExampleActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
   }
}
```

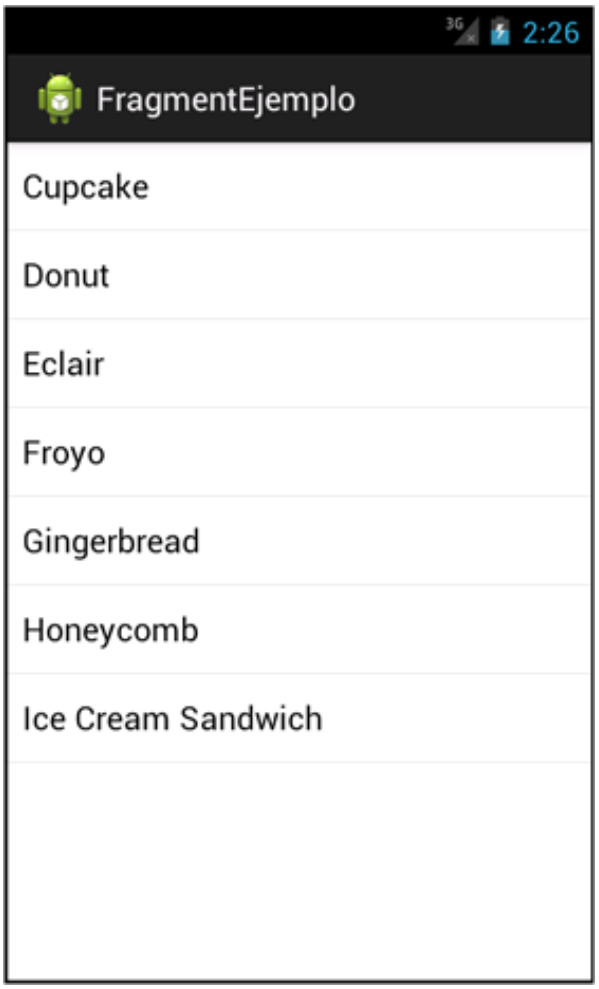
Y el archivo de manifiesto que contiene las diferentes declaraciones:

```
<application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
           android:name=".FragmentExampleActivity"
           android:label="@string/app_name" >
           <intent-filter>
             <action android:name="android.intent.action.MAIN" />
```

```
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DetailActivity"></activity>
    </application>
```

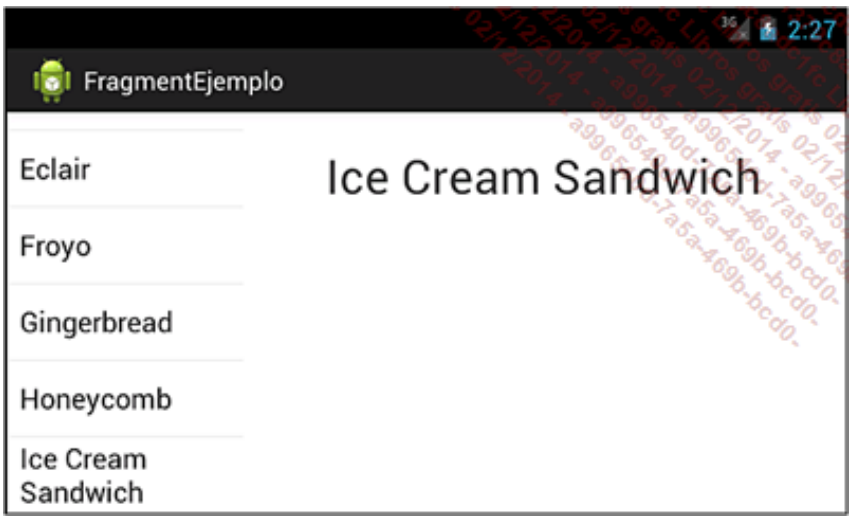
Ahora puede probar el ejemplo, empezando en modo vertical y pasando después al modo apaisado.

Con lo que obtendrá los siguientes resultados:



Modo vertical - Lista de versiones de Android

Modo vertical - Detalle de una versión de Android



Modo apaisado

Paso a modo de pantalla completa

Algunas aplicaciones requieren pasar a pantalla completa (reproducción de vídeo, por ejemplo). Un paso a modo de pantalla completa puede significar que la aplicación oculta la barra de notificación del teléfono y/o la barra de navegación (excepto en el caso de una tablet).

El caso de una tablet es bastante diferente debido a que las barras de notificación y de navegación se encuentran ambas en la barra de sistema. Lo que no permite ocultarlas pero sí atenuarlas para hacerlas menos visibles.

A continuación se muestra el método que permite ocultar la barra de notificaciones en un smartphone y atenuarla en una tablet:

```
onSystemUiVisibilityChange(int visibility)
```

Este método recibe un parámetro que puede adquirir los siguientes valores:

- **SYSTEM_UI_FLAG_LOW_PROFILE**: permite atenuar la barra de navegación.
- **SYSTEM_UI_FLAG_HIDE_NAVIGATION**: permite ocultar la barra de navegación.


También puede atenuar la barra de navegación en un smartphone.

Sin embargo, Android 4 introduce una limitación (debido a la ausencia de botón físico): a la menor interacción del usuario con la actividad actual, las barras de notificación aparecen de nuevo.

Si lo que realmente desea es ocultar la barra de notificación de un smartphone (en reproducción de video o videojuegos, por ejemplo), basta con utilizar el siguiente método en la inicialización de la actividad:

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN);
```


No oculte la barra de notificaciones a no ser que sea necesario, ya que es importante para la interacción del usuario con el dispositivo (acceso a los eventos importantes que se producen en el dispositivo).

Interfaces dinámicas

Es posible crear interfaces más enriquecidas mediante la creación de interfaces dinámicas. Una interfaz dinámica es una interfaz creada en Java, directamente en el archivo fuente de una actividad.

Se puede combinar una interfaz dinámica con una interfaz estática (archivo XML).

La creación dinámica de interfaces es muy útil en el caso en que desee añadir componentes al vuelo en una interfaz.

Para ilustrar esta funcionalidad, el siguiente ejemplo crea una interfaz compuesta de un botón que servirá para añadir campos de texto (**EditText**) dinámicamente en la misma.

Esta vista contiene, simplemente, un **LinearLayout** y un botón que tiene un identificador.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="vertical"
      android:id="@+id/linearlayout">

      <Button
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/addEditText"
            android:id="@+id/addBtn"/>

</LinearLayout>
```

Con el clic del botón, los campos de texto editables se añadirán dinámicamente a la vista.

```
public class DynamicViewActivity extends Activity {

    @Override
      public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);
          final LinearLayout linearLayout = (LinearLayout)
findViewById(R.id.linearlayout);
          Button btn = (Button) findViewById(R.id.addBtn);
          btn.setOnClickListener(new OnClickListener() {
               @Override
               public void onClick(View v) {
                    EditText edit = new
EditText(DynamicViewActivity.this);
                    edit.setHint(R.string.newEditText);
                    LayoutParams layoutParams = new
LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.WRAP_CONTENT);
            edit.setLayoutParams(layoutParams);
            linearLayout.addView(edit);
         }
      });
    }
}
```

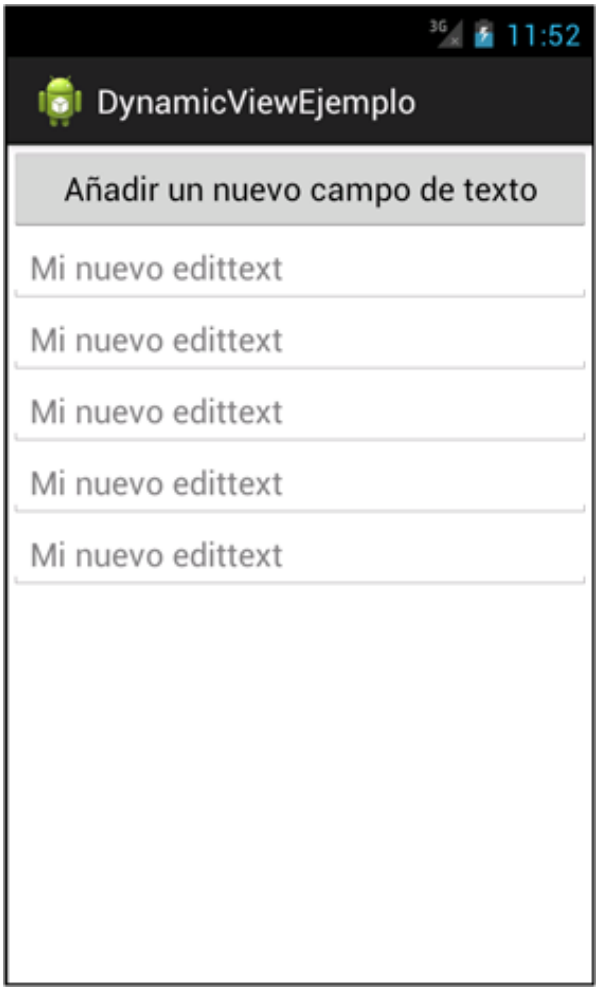
Para crear dinámicamente elementos **EditText**:

- Hay que crear el campo de edición mediante su constructor pasándole el contexto necesario

para su inicialización.

- Este campo de texto tiene un texto por defecto (**setHint**).
- Los **layoutParams** sirven para especificar valores y propiedades a un **EditText** (especialmente su anchura y altura).
- Para finalizar, hay que añadir el campo de texto editable que se acaba de crear en el**LinearLayout** obtenido anteriormente (mediante su identificador).

A continuación se muestra el resultado obtenido:



Creación de pestañas

1. Principio

Las pestañas permiten cambiar de vista fácilmente y mejorar, de este modo, la experiencia de usuario de una aplicación.

Existen tres tipos de pestañas:

- **Pestañas scrollables** (por ejemplo, las que se usa en Google Play):
  - Pueden contener un gran número de pestañas.
  - La navegación entre pestañas se realiza deslizando un dedo de izquierda a derecha o en sentido inverso.

- **Pestañas fijas** (por ejemplo, las de la aplicación Contactos):
  - Las pestañas se muestran unas al lado de las otras.
  - Puede haber un máximo de tres pestañas.

- **Pestañas apiladas** (por ejemplo, las de la aplicación Youtube):
  - Permiten separar o fusionar pestañas con una **ActionBar** (véase el capítulo Creación de interfaces sencillas - ActionBar).

2. Implementación de pestañas scrollables

Este componente se puede usar mediante la clase **ViewPager**, disponible en el **Compatibility Package**. Este último se puede descargar en la sección **extras** del SDK Android (véase el capítulo El entorno de desarrollo - SDK Android).

Una vez descargado, debe integrar la librería descargada en el proyecto mediante el archivo jar que se encuentra en la carpeta **extras** de su SDK Android.

El primer paso consiste en integrar el **ViewPager** en una interfaz XML.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:background="#fff" >

    <android.support.v4.view.ViewPager
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/viewPager"/>

</LinearLayout>
```

Puede comprobar que el **ViewPager** no es un elemento nativo del SDK sino que proviene del**Compatibility Package**.

El próximo paso consiste en crear un adapter personalizado para la clase **ViewPager**. Este adapter debe heredar de la clase **PagerAdapter** e implementar los siguientes métodos:

- **getCount**: devuelve el número de vistas disponibles en la interfaz.
- **instantiateItem**: crea la página situada en una posición determinada.
- **destroyItem**: elimina una página situada en una posición determinada.
- **isViewFromObject**: determina si la vista está asociada a un objeto.
- **finishUpdate**: se invoca cuando se han realizado todas las actualizaciones de la página actual.
- **restoreState**: restaura el estado de una página.
- **saveState**: guarda el estado de una página.
- **startUpdate**: se invoca para actualizar la página actual.

```
private class ViewPagerAdapter extends PagerAdapter {

   @Override
   public int getCount() {
        return NUMBER_OF_PAGES;
   }

   @Override
   public Object instantiateItem(View collection, int position)  {
   TextView tv = new TextView(ViewPagerExempleActivity.this);
   tv.setText("Pág. número: " + position);
   tv.setTextColor(Color.BLACK);
   tv.setTextSize(30);

   ((ViewPager) collection).addView(tv, 0);
   return tv;
}

   @Override
   public void destroyItem(View collection, int position,
Object view) {
           ((ViewPager) collection).removeView((TextView) view);
         }



   @Override
   public boolean isViewFromObject(View view, Object object) {
          return view==((TextView)object);
   }

   @Override
   public void finishUpdate(View arg0) {}


   @Override
   public void restoreState(Parcelable arg0, ClassLoader arg1) {}
```

```
    @Override
    public Parcelable saveState() {
                return null;
    }

    @Override
    public void startUpdate(View arg0) {}

}
```

A continuación se muestran algunas explicaciones sobre el adapter que se acaba de crear:

- El método **getCount** debe devolver el número de páginas disponibles en el adapter.
- **instantiateItem**: permite instanciar la vista actual. En el ejemplo, se crea un campo de texto en el que hay que mostrar el número de página. Después, este campo de texto se añade al**ViewPager**.
- **destroyItem**: permite destruir el elemento actual.
- **isViewFromObject**: compara la vista actual con el objeto que se recibe como parámetro.

Con lo que se obtendrá:

Popups

1. Toasts

Los toasts sirven para mostrar un mensaje corto, una indicación que no necesita la interacción del usuario.

Es posible crear un toast desde una actividad, un servicio o cualquier otra clase que tenga un contexto.

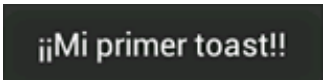
```
Toast.makeText(ToastExampleActivity.this, R.string.toast,
Toast.LENGTH_SHORT).show();
```

Para crear un toast, utilice el método **makeText** que recibe como parámetros:

- El contexto.
- La cadena que se mostrará.
- La duración de la visualización del toast.
  - Dispone de dos valores predefinidos: **Toast.LENGTH_SHORT** y **Toast.LENGTH_LONG**.

El método **show** sirve para mostrar el toast.

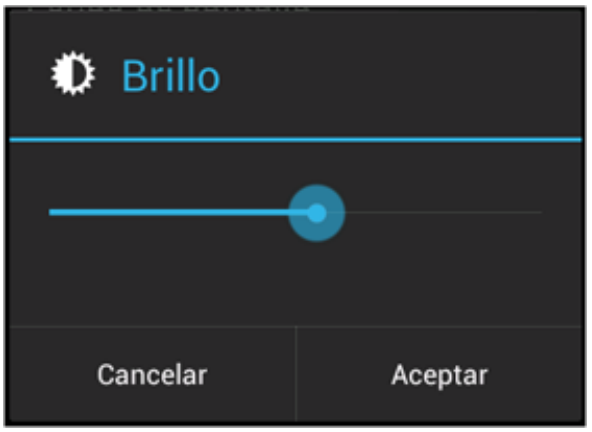
Con lo que se obtendrá:



No olvide llamar al método **show** para mostrar el **Toast**.

2. AlertDialog

Un **AlertDialog** es un cuadro de diálogo que permite mostrar un mensaje y realizar una interacción con el usuario.



Se compone de tres partes:

- Un **título (opcional)**: representa el título del cuadro de diálogo.
- Un **contenido**: representa el contenido de un cuadro de diálogo. Puede contener campos de texto, sliders, casillas de activación, radiobuttons, etc.

- Un **botón o varios botones de acción**: permite al usuario elegir la opción que se realizará.

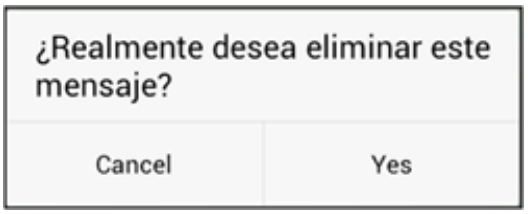
A continuación se muestra un ejemplo de un cuadro de diálogo que solicita al usuario la confirmación de la eliminación de un mensaje.

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage(R.string.alert_dialog_msg).setCancelable(false
).setPositiveButton("Yes", new OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
         dialog.cancel();
    }
}).setNegativeButton("Cancel", new OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
         dialog.cancel();
    }
});
builder.create().show();
```

El código anterior:

- Utiliza un **AlertDialog.Builder** para crear el cuadro de diálogo.
- Especifica el mensaje así como los botones que se mostrarán. También puede tener un único botón mediante el método **setNeutralButton**.
- Después, se invoca al método **create** para crear el cuadro de díalogo.
- Finalmente, utiliza el método **show** para mostrar el cuadro de diálogo.

Lo que generará:



3. ProgressDialog

Las barras de progreso sirven para informar a un usuario acerca del grado de avance de una tarea.

Existen dos tipos de barras de progreso:

- **Barra de progreso acotada**: si desea, por ejemplo, conocer el porcentaje de avance de una tarea (por ejemplo, la descarga de una aplicación a través de Google Play).

- **Barra de progreso sin acotar**: si la duración del tratamiento no se puede calcular, puede usar este tipo de barra de progreso:

a. Implementación

Para implementar una barra de progreso, hay que usar la clase **ProgressDialog**.

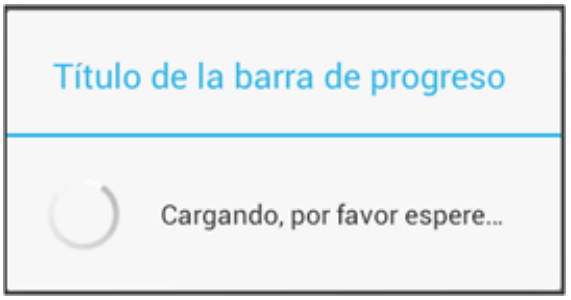
A continuación se muestra un ejemplo de creación de una barra de progreso:

```
ProgressDialog dialog = ProgressDialog.show(
ProgressDialogExampleActivity.this,
getResources().getString(R.string.dialog_title), getResources()
.getString(R.string.dialog_load), true);
```

La construcción requiere:

- Un contexto.
- Una cadena de caracteres que sirva de título en el cuadro de diálogo de progreso (opcional).
- Una cadena de caracteres que represente el mensaje que aparece en el cuadro de diálogo de progreso.

Lo que generará:



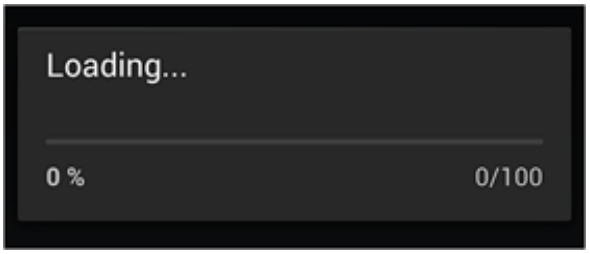
Una vez realizado el tratamiento, deberá detener la barra de progreso mediante el método**dismiss**:

```
dialog.dismiss();
```

Para crear una barra de progreso horizontal, hay que:

- Crear una instancia de la clase **ProgressDialog**.
- Definir el estilo de la barra de progreso (estilo horizontal).
- Definir el mensaje.
- Mostrar la barra de progreso.

```
ProgressDialog progressDialog;
progressDialog = new
ProgressDialog(ProgressDialogExampleActivity.this);
progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setMessage("Loading...");
progressDialog.show();
```



Para actualizar el avance de la tarea en curso, utilice el método **setProgress**:

```
progressDialog.setProgress(PROGRESS_VALUE);
```

4. Cuadro de diálogo personalizado

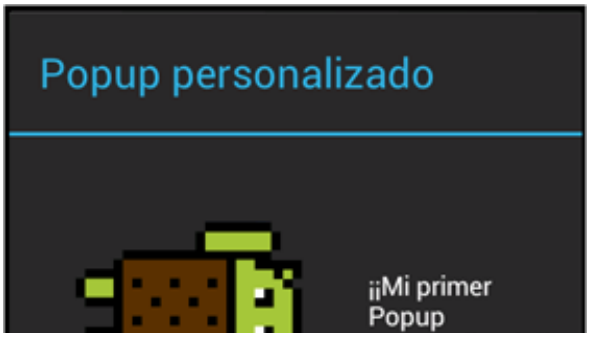
Puede crear cuadros de diálogo personalizados con un layout específico. Para ello, cree un archivo XML que represente el layout personalizado.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                    android:id="@+id/layout_root"
                    android:orientation="horizontal"
                    android:layout_width="fill_parent"
                    android:layout_height="fill_parent"
                    android:padding="10dp"
                    >
      <ImageView android:id="@+id/alert_img"
                    android:layout_width="wrap_content"
                    android:layout_height="fill_parent"
                    android:layout_marginRight="10dp"
                    />
      <TextView android:id="@+id/alert_msg"
                    android:layout_width="wrap_content"
                    android:layout_height="fill_parent"
                    android:textColor="#FFF"
                    android:gravity="center_vertical"
                    />
</LinearLayout>
```

Después, en la actividad, hay que:

- Crear el cuadro de diálogo.
- Asociar el cuadro de diálogo con el archivo XML que representa la vista personalizada.
- Especificar el título, una imagen y una descripción para el cuadro de diálogo.

```
Dialog dialog = new
Dialog(CustomAlertDialogExampleActivity.this);

dialog.setContentView(R.layout.custom_alert);
dialog.setTitle("Popup Personalizado")

TextView text = (TextView) dialog.findViewById(R.id.alert_msg);
text.setText("¡¡Mi primer Popup personalizado!! ");
ImageView image = (ImageView)dialog.findViewById(R.id.alert_img);
image.setImageResource(R.drawable.nyan);

dialog.show();
```


Popup personalizado
¡¡Mi primer
Popup

Preferencias

Las preferencias permiten crear pantallas específicas dedicadas a la creación y gestión de las preferencias de un usuario en una aplicación o, de forma global, en el dispositivo.

Desde la versión 3.0 de Android, debe utilizar la clase **PreferenceFragment** para crear las pantallas de preferencias de una aplicación.

Las preferencias creadas mediante la clase **PreferenceFragment** se guardarán automáticamente en las **SharedPreferences** (véase el capítulo Persistencia de datos - SharedPreferences).

Para obtener las **SharedPreferences**, invoque al método **getDefaultSharedPreferences**.

Para crear una pantalla de preferencias, dispone de varios componentes. Cada componente tiene su contexto de uso:

- **Casilla de activación/Switcher**: utilice este componente para las opciones que sólo tengan dos estados (activo o inactivo).
- **Selección múltiple**: utilice este componente cuando el usuario deba elegir una opción entre varias posibles.
- **Slider**: utilice este componente cuando el usuario deba elegir un valor entre un rango de valores posibles.

Para crear una pantalla que represente las preferencias de una aplicación, comience creando una carpeta llamada **xml** en los recursos de la aplicación. En efecto, la carpeta **xml** es la ubicación natural de las interfaces de preferencias.

Cree un archivo que represente una interfaz de preferencias:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android" >

     <PreferenceCategory android:title="@string/notification" >
         <SwitchPreference
             android:key="checkbox_preference"
             android:title="@string/enable_notification" />
     </PreferenceCategory>
     <PreferenceCategory android:title="@string/data" >
         <CheckBoxPreference
             android:summary="@string/save_data_summary"
             android:title="@string/save_data_title" />
      </PreferenceCategory>

</PreferenceScreen>
```

Una pantalla que representa una interfaz de preferencias siempre comienza con la etiqueta**PreferenceScreen**.

Puede anidar varias etiquetas **PreferenceScreen** para obtener varias pantallas de preferencias anidadas.

Cada sección distinta de una interfaz de preferencias empieza con la etiqueta **PreferenceCategory**. Cada categoría puede contener uno o varios elementos.

El ejemplo anterior tiene dos categorías:

- La primera contiene un **Switcher**.

- La segunda contiene una casilla de activación.

Todos los elementos que pueden formar parte de una pantalla de preferencias tienen un nombre que termina por **Preference** (SwitchPreference, CheckBoxPreference...).

Modifique la actividad principal para integrar el fragment que representa la pantalla de preferencias:

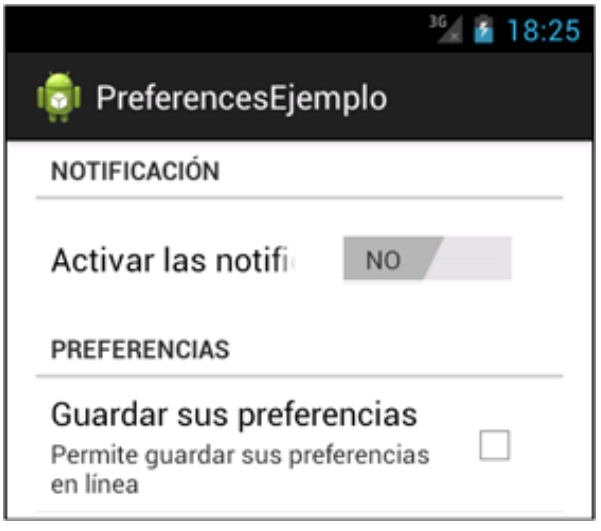
```
@Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
getFragmentManager().beginTransaction().replace(android.
R.id.content, new MyPreferenceFragment()).commit();
      }

    public static class MyPreferenceFragment extends
PreferenceFragment {
            @Override
            public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                addPreferencesFromResource(R.xml.preference_screen);
            }
      }
```

En el método **onCreate**, la inserción del fragment se realiza siguiendo los siguientes pasos:

- Se obtiene una instancia de la clase **FragmentManager**, para interactuar con fragments.
- Se invoca al método **beginTransaction**, que permite comenzar una serie de operaciones con un fragment.
- Se invoca al método **replace**, que permite integrar la clase personalizada (**MyPreferenceFragment**) en la interfaz como fragment principal.
- Se invoca al método **commit**, que valida todas modificaciones realizadas en los pasos anteriores.

Después, cree una clase que herede de la clase **PreferenceFragment** y añada el fragment declarado en el archivo XML.

Con lo que obtendrá:



WebView

El framework Android permite, gracias a la clase **WebView**, incluir páginas HTML en el interior de una aplicación.

Esta clase utiliza el **WebKit** de Android para mostrar páginas HTML, el historial, tratar código JavaScript, hacer zoom, etc.

Puede mostrar una página web remota, una página almacenada de manera local en el dispositivo o, simplemente, incluir código HTML.

1. Ejemplo de una página web remota

El primer paso consiste en crear un archivo XML que represente la vista e incluir el componente**WebView**.

```
<?xml version="1.0" encoding="utf-8"?>
<WebView
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/webview"
android:layout_width="fill_parent"
android:layout_height="fill_parent" />
```

A continuación, hay que crear una actividad que permita asociar la vista declarada anteriormente con la actividad y especificar la URL que el **WebView** debe cargar.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    WebView webView = (WebView) findViewById(R.id.webview);
    webView.loadUrl("http://www.tutos-android.com");
}
```

Puede especificar la URL mediante el método **loadUrl**, accesible desde la instancia de la clase WebView.

Sin olvidar añadir la *permission* de acceso a Internet en el manifiesto de la aplicación:

```
<uses-permission android:name="android.permission.INTERNET" />
```

2. Ajustes del WebView

Para configurar un **WebView**, puede acceder a los ajustes mediante el método **getSettings**.

A continuación se muestra un ejemplo que obtiene los parámetros de un WebView para mostrar los botones de zoom (**setBuiltInZoomControls**) y permitir la ejecución de JavaScript (**setJavascriptEnabled**).

```
WebSettings settings = webView.getSettings();
settings.setBuiltInZoomControls(true);
settings.setJavaScriptEnabled(true);
```

3. Gestión del botón retorno

Debe sobrecargar la gestión del botón retorno para simular el comportamiento del botón de retorno de un navegador web en las páginas web en una aplicación Android.

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if ((keyCode == KeyEvent.KEYCODE_BACK) &&
mWebView.canGoBack()) {
        mWebView.goBack();
        return true;
      }
      return super.onKeyDown(keyCode, event);
}
```

- Para ello, sobrecargue el método ***onKeyDown***. Compruebe que el clic se corresponde con el botón de retorno y que la navegación no se encuentra en la primera página (que sea posible acceder a una página anterior).

4. Uso de Android nativo en JavaScript

Puede utilizar código JavaScript en los **WebView**, e incluso combinarlo con código nativo.

El objetivo del ejemplo siguiente es mostrar un toast Android mediante código JavaScript.

- Para comenzar, cree una clase que represente la interfaz JavaScript. Existe un método que permite mostrar un Toast.

```
public class JavaScriptToastInterface {
   Context context;

   JavaScriptToastInterface(Context c) {
      context = c;
   }

   public void showToast(String toastMsg) {
      Toast.makeText(context, toastMsg,
Toast.LENGTH_SHORT).show();
    }
}
```

- A continuación, active JavaScript en el **WebView** y asocie la interfaz JavaScript al mismo mediante el método **addJavascriptInterface**.

El segundo parámetro del método se corresponde con el identificador que tendrá esta interfaz en el código HTML. Esto le permite añadir más interfaces JavaScript que tengan identificadores distintos.

```
WebView webView = (WebView) findViewById(R.id.webview);
WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);
webView.addJavascriptInterface(new
JavaScriptToastInterface(this), "Android");
webView.loadData(WEB_CONTENT, "text/html", "UTF-8");
```

Se va a necesitar un contenido web para mostrar:

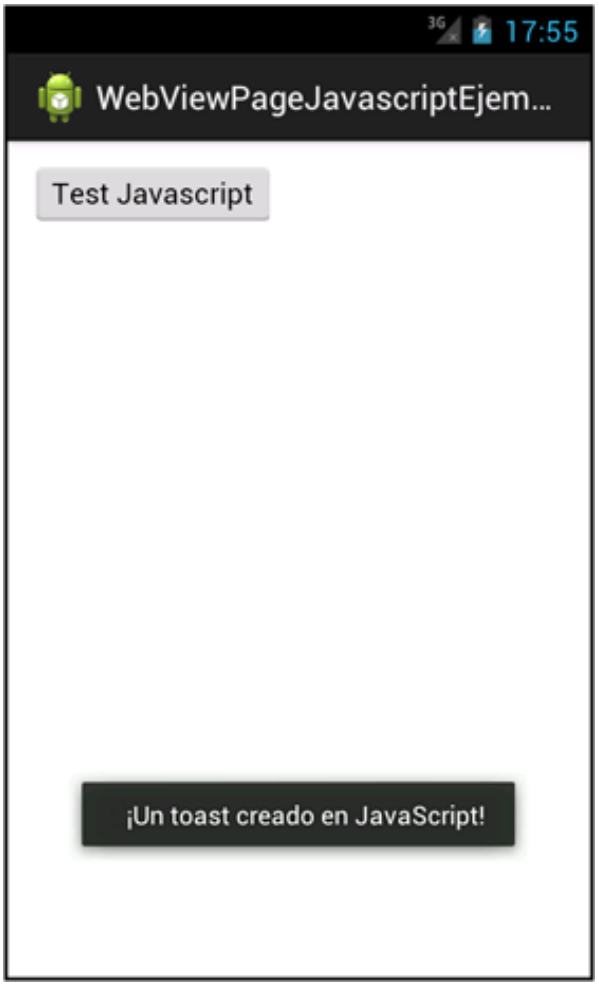
```
<input type="button" value="Test Javascript"
onClick="showAndroidToast('¡Un Toast creado en JavaScript!')" />

<script type="text/javascript">
     function showAndroidToast(toast) {
         Android.showToast(toast);
     }
</script>
```

Este fragmento de código representa un botón que ejecuta el método **showAndroid** presente en la interfaz JavaScript.

- Almacene este código HTML en una variable de tipo **String** y cárguela en el WebView mediante el método **loadData**.

```
String WEB_CONTENT = "<input type=\"button\" value=\"Test
Javascript\"  onClick=\"showAndroidToast(\'¡Un toast creado
en JavaScript!\')\" /><script  type=\"text/javascript\">function
showAndroidToast(toast) {Android.showToast(toast);}</script>";
```

Obtendrá:



Buenas prácticas

A continuación se muestran algunas recomendaciones que debe seguir para que sus aplicaciones se adapten con éxito a las diferentes situaciones que suelen aparecer en el desarrollo de aplicaciones:

1. Mantenerse independiente de la resolución de pantalla

- En su aplicación, utilice los **dp** (**independent pixel**) para declarar los tamaños de los distintos elementos y componentes así como los **sp** (**independent scale**) para los tamaños de las fuentes.
- Cree carpetas **drawable** para cada resolución (véase el capítulo Creación de interfaces sencillas - Recursos).

2. Mantenerse independiente del tamaño de pantalla

- Priorice el uso de los tamaños predefinidos (**wrap_content** / **match_parent**) para diferentes tamaños de elementos, lo que permitirá a las vistas adaptarse a los distintos tamaños de pantalla.
- Utilice el **RelativeLayout**, que permite controlar con mayor precisión la ubicación de los distintos elementos de una vista, colocando los unos en relación a los otros y no en función del tamaño de pantalla.
- Cree layouts para los distintos tamaños de pantalla (small, normal, large, xlarge…) y las posiciones de pantalla (vertical y apaisado).
- Especifique los tamaños de pantalla compatibles mediante la etiqueta **supports-screens**(véase el capítulo Principios de programación - Manifiesto).
- Utilice imágenes estirables (herramienta **9-patch**). La herramienta **draw9patch**, incluida con el SDK Android (carpeta **tools**), permite especificar la forma de cómo se deberá estirar una imagen según el caso mediante puntos extensibles definidos en la misma.

3. Ser independiente de la versión de Android utilizada

Algunas APIs, funcionalidades o componentes sólo están incluidas en algunas versiones de Android. Tenga la precaución de probar la versión actual en el dispositivo antes de utilizar una funcionalidad que pueda no estar disponible en esta versión del sistema operativo.

```
if (Build.VERSION.SDK_INT <
Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
        //Dispositivo versión 4.0.0 o superior
} else {
    //Dipositivo anterior a 4.0.0
}
```

Cree las carpetas específicas para sus interfaces si desea personalizar una interfaz en función de las versiones de Android (menu - v14 / values - v11 /layout - v14...).

4. Ser eficiente

- No cree objetos inútilmente.
- Las constantes deben ser estáticas (la palabra clave **final** permite indicar que el valor almacenado en una variable no puede ser modificado).

```
private static final my_const = 42;
```

- Evite el uso de Getters/Setters, acceda directamente a los atributos de la clase si es posible.
- No utilice cualquier librería en una aplicación Android.

Optimizar sus interfaces

En cada creación de una actividad, se carga el layout asociado a esta actividad (método**setContentView** o **inflate**). Esta carga puede ser muy costosa si la interfaz en cuestión está mal diseñada o es muy pesada. Por este motivo Android proporciona muchos mecanismos para optimizar las distintas interfaces que componen su aplicación.

1. Inspeccionar la jerarquía de sus interfaces

El primer método para optimizar interfaces Android consiste en inspeccionar la jerarquía de la vista en cuestión. Para ello, el SDK Android proporciona una herramienta llamada **HierarchyViewer** (se encuentra en la carpeta **tools** del SDK).

Para utilizar esta herramienta:

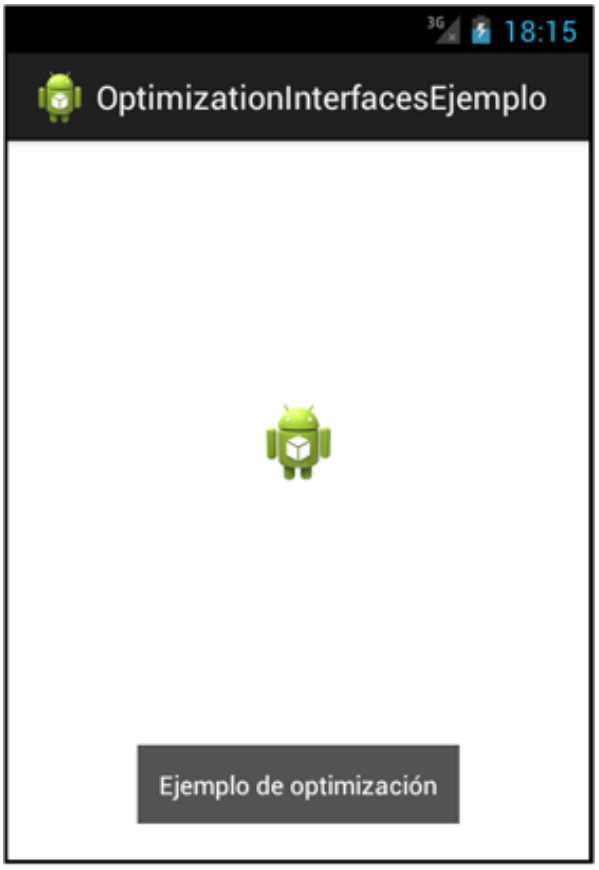
- Inicie la aplicación en un emulador.
- Muestre la actividad deseada (la que desea inspeccionar).
- Arranque el **HierarchyViewer**.
- Seleccione el proceso correspondiente a la aplicación en cuestión y, a continuación, haga clic en**Load View Hierarchy**.

Se mostrará un diagrama de la interfaz.

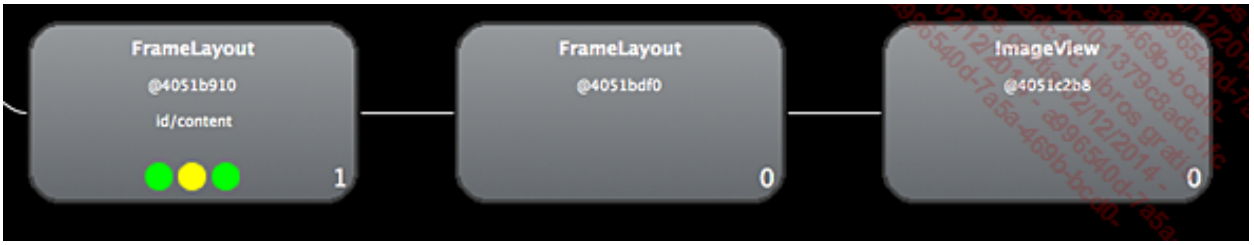
A continuación se muestra un ejemplo sencillo de un **FrameLayout** que contiene una imagen y un texto.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:src="@drawable/ic_launcher" />

</FrameLayout>
```

Se generará la siguiente vista:



La ejecución de la herramienta **HierarchyViewer** sobre la vista declarada anteriormente genera el siguiente gráfico:



Puede observar que el **FrameLayout** que se ha declarado en la vista sigue a otro **FrameLayout**(añadido automáticamente por Android tras la creación de una vista).

La conclusión es que el **FrameLayout** que se ha declarado es inútil y puede utilizar la etiqueta**<merge>** para fusionar la vista (sin **FrameLayout**) con el **FrameLayout** declarado automáticamente (véase la sección Fusionar layouts). Con ello, se optimiza la vista evitando la carga de un layout innecesario.

2. Fusionar layouts

Android ofrece el tag **<merge>** para reducir y optimizar el número de niveles de una interfaz (véase el ejemplo anterior). Esta optimización se realiza fusionando los componentes declarados tras el**merge** con el layout situado encima del **merge**.

- Aplique dicha optimización al ejemplo anterior:

```
<merge
xmlns:android="http://schemas.android.com/apk/res/android">

    <ImageView
        android:layout_width="fill_parent"
```

```
        android:layout_height="fill_parent"
        android:scaleType="center"
        android:src="@drawable/ic_launcher" />

</merge>
```

La etiqueta **merge** presenta algunas limitaciones:

- Los layouts fusionados deben ser idénticos o tener el mismo comportamiento.
- La etiqueta sólo puede usarse en la raíz de un archivo XML.
- Cuando se utiliza la etiqueta **merge**, hay que especificar el layout padre con el que se asociará la vista y definir el parámetro **attachToRoot** a verdadero.

3. Incluir vistas

Puede modularizar las interfaces para incluir y reutilizar vistas ya creadas en otras vistas. Gracias a la etiqueta **include** que proporciona Android es posible usar esta funcionalidad.

Cuando la utilice, debe especificar:

- Un identificador del layout incluido: para poder inicializarlo en la actividad y recuperar su contenido.
- El layout que se incluirá.

El siguiente ejemplo incluye en un **LinearLayout** otro layout declarado en otro archivo.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <include
        android:id="@+id/included_layout"
        layout="@layout/main" />

    <Button
        android:id="@+id/btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```

4. Carga perezosa (Lazy Loading) de layouts

La carga de una vista es muy costosa y, cuanto más elementos contiene, mayor es la duración. Para minimizar el número de vistas cargadas en la creación de una interfaz puede utilizar la clase**ViewStub**.

Un **ViewStub** utiliza un mecanismo de carga perezosa. Cualquier elemento de la interfaz que se haya declarado mediante un **ViewStub** sólo se cargará cuando se haga visible.

A continuación se muestra un ejemplo de una interfaz que contiene dos botones. El segundo se declara en **ViewStub** y sólo se muestra tras hacer clic en el primer botón.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/enable"
        android:text="@string/enable_view_stub" />

    <ViewStub
        android:id="@+id/view_stub"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inflatedId="@+id/view_stub_visible"
        android:layout="@layout/view_stub_btn" />

</LinearLayout>
```

De este modo, la declaración de un **ViewStub** se compone de:

- Un identificador.
- Un identificador cuando el **ViewStub** se haga visible.
- El layout que el **ViewStub** cargará.

A continuación, declare el layout especificado por el **ViewStub**:

```
<?xml version="1.0" encoding="utf-8"?>
<Button
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/btn_stub" />
```

Para finalizar, hay que obtener el **ViewStub** y hacerlo visible tras hacer clic en el primer botón:

```
final View stub = findViewById(R.id.view_stub);

Button enable = (Button) findViewById(R.id.enable);
enable.setOnClickListener(new OnClickListener() {
   @Override
   public void onClick(View v) {
      stub.setVisibility(View.VISIBLE);
   }
});
```

Una vista tiene tres tipos de visibilidad:

VISIBLE: la vista es visible.

INVISIBLE: la vista es invisible pero ocupa siempre el espacio que se le ha asignado.

GONE: la vista es invisible y no ocupa ni el espacio que se le ha asignado.

Introducción

El framework Android le permite almacenar los datos necesarios para el buen funcionamiento de una aplicación de varias formas. Todo depende de las características de almacenamiento que usted necesite.

Podrá elegir entre los siguientes tipos de almacenamiento:

- **SharedPreferences**: permite almacenar datos en forma de pares clave/valor.
- **Archivos**: permite almacenar datos en archivos (creados en la memoria interna o externa del dispositivo).
- **Almacenamiento en base de datos**: permite crear una base de datos SQLite para almacenar los datos necesarios para el funcionamiento de la aplicación.

SharedPreferences

La clase **SharedPreferences** proporciona un conjunto de métodos que permiten almacenar y recuperar muy fácilmente un par **clave/valor**, con la particularidad de que solo contiene datos primitivos (boolean, float, int, string, Parcelable...).

Estos datos persisten incluso cuando la aplicación no se está ejecutando.

El primer paso para el uso de datos **SharedPreferences** consiste en recuperar una instancia de esta clase. Para ello, dispone de dos métodos:

- **getPreferences(int mode)**: utilice este método si necesita un único archivo de preferencias en su aplicación, ya que este método no le permite especificar el nombre del archivo donde se almacenarán los datos **SharedPreferences**.
- **getSharedPreferences(String name, int mode)**: utilice este método si necesita varios archivos de preferencias que podrá identificar por su nombre.

El parámetro **mode** puede adquirir uno de los siguientes valores:

- **MODE_PRIVATE**: el archivo es privado y, por lo tanto, está reservado para su aplicación (valor por defecto).
- **MODE_WORLD_READABLE**: el archivo es accesible en modo lectura para otras aplicaciones.
- **MODE_WORLD_WRITABLE**: el archivo es accesible en modo escritura para otras aplicaciones.
- **MODE_MULTI_PROCESS**: permite que el archivo se pueda usar simultáneamente por varios procesos (disponible a partir de Gingerbread - Android 2.3).

```
SharedPreferences myPref = getPreferences(MODE_PRIVATE);
boolean myValue = myPref.getBoolean(SHARED_KEY, false);
```

Una vez se ha obtenido la instancia de la clase **SharedPreferences**, puede utilizar los diferentes métodos disponibles para obtener los datos almacenados. Esta recuperación se realiza mediante una clave que sirve de identificador del dato solicitado. También puede especificar un valor por defecto si el dato no existiera en el **SharedPreferences** consultado.

Para poder añadir datos a un **SharedPreferences**, hay que obtener una instancia de la clase **Editor**. Para ello, utilice el método **edit**, disponible a través la instancia **SharedPreferences**.

```
SharedPreferences.Editor editor = myPref.edit() ;
```

A continuación, puede utilizar los distintos métodos disponibles para agregar valores a la colección de tipo **SharedPreferences**.

```
editor.putBoolean(SHARED_KEY, true) ;
```

Su pare **clave/valor** no se agregará de manera efectiva hasta la llamada al método **commit**.

```
editor.commit();
```

Puede añadir varios valores en una colección SharedPreferences antes de invocar al método commit (buena práctica).

Puede actualizar valores ya almacenados en la colección **SharedPreferences** de una aplicación de forma sencilla, obteniendo una variable de tipo **Editor**. Gracias a su clave, podrá realizar las modificaciones.

```
SharedPreferences myPref = getPreferences(MODE_PRIVATE);
SharedPreferences.Editor editor = myPref.edit();
```

```
editor.putBoolean(SHARED_KEY, false);
editor.commit();
```

Para eliminar un valor de una colección **SharedPreferences**, utilice el método **remove** presente en la clase **Editor**.

```
editor.remove(SHARED_KEY);
```

También puede hacer que una aplicación se suscriba a las modificaciones realizadas en una colección**SharedPreferences** utilizando un listener en la colección. Para ello, utilice el método**registerOnSharedPreferenceChangeListener**.

```
myPref.registerOnSharedPreferenceChangeListener(new
OnSharedPreferenceChangeListener() {
   @Override
   public void onSharedPreferenceChanged(SharedPreferences
sharedPreferences, String key) {
      //Verificar la colección SharedPreferences y la clave usada para realizar la modificación
   }
});
```

Utilice el método **unregisterOnSharedPreferenceChangeListener** para anular la suscripción a un **SharedPreferences**.

Almacenamiento interno

Puede guardar archivos directamente en la memoria interna del teléfono. Por defecto, los archivos guardados por una aplicación no son accesibles para el resto de aplicaciones.

Si el usuario desinstala una aplicación, los archivos correspondientes, albergados en el almacenamiento interno, se eliminarán.

1. Escritura de un archivo

A continuación se muestra un ejemplo de creación de un archivo en el almacenamiento interno del teléfono.

```
String FILENAME = "miArchivo.txt";
String str = "Esto es un ejemplo de almacenamiento interno";

FileOutputStream fos;
try {
    fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
    fos.write(str.getBytes());
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
} finally {
     fos.close();
}
```

La creación de un archivo en el almacenamiento interno se realiza siguiendo los siguientes pasos:

- Invoque al método **openFileOutput** con el nombre del archivo y el modo de apertura del archivo (privado, público en modo lectura, público en modo escritura, en adición).
- Utilice el método **write** para escribir los datos en el archivo.
- No se olvide de cerrar el archivo para las operaciones de escritura.

Estas operaciones pueden producir dos excepciones: si no se puede encontrar el archivo o si se producen problemas en la escritura (un problema de permisos, por ejemplo).

2. Lectura de un archivo

La lectura de un archivo se realiza del mismo modo. La diferencia está en el uso de los métodos**openFileInput** y **read**.

```
FileInputStream fis;
try {
   fis = openFileInput(FILENAME);
   byte[] buffer = new byte[1024];
   StringBuilder content = new StringBuilder();
   while ((fis.read(buffer)) != -1) {
          content.append(new String(buffer));
   }
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
```

```
} finally {
    fis.close()
}
```

- La variable **content** alberga el contenido del archivo.
- El buffer se va rellenando a medida que se van haciendo lecturas.
- El método **read** devuelve -1 cuando acaba la lectura del archivo.

Estas operaciones pueden producir dos tipos de excepciones: si no se puede encontrar el archivo o si se producen problemas de lectura.

Los tratamientos de lectura/escritura en un archivo deben realizarse en un Thread diferente al del UI Thread (véase el capítulo Tratamiento en tareas en segundo plano).

a. Utilización de archivos en caché

Si desea almacenar datos temporales sin tener, por ello, que guardarlos en archivos persistentes, puede utilizar el método **getCacheDir()** para abrir una carpeta de caché.

Alojada en la memoria del teléfono, esta carpeta representa la ubicación que servirá para guardar los archivos temporales de su aplicación.

Si el dispositivo tiene problemas de espacio, Android podrá eliminar estos archivos para obtener más espacio.

Sus archivos de caché deben ocupar un espacio acotado en la memoria del teléfono (1 MB como máximo, por ejemplo).

Recuerde que, en la desinstalación de la aplicación, la carpeta de caché se eliminará.

Almacenamiento externo

1. Comprobar la disponibilidad del almacenamiento externo

Para evitar problemas con la memoria disponible, puede almacenar los archivos de una aplicación en la memoria externa del dispositivo.

Los tratamientos de lectura/escritura en un archivo deben realizarse en un Thread diferente al UI Thread.

Hay dos tipos de memoria externa:

- Almacenamiento externo desmontable (tarjeta SD, por ejemplo).
- Almacenamiento externo no desmontable (alojado en la memoria física del teléfono).

Los archivos creados en el almacenamiento externo se abren en modo lectura para el usuario y el resto de aplicaciones. Además, el usuario podrá eliminar estos archivos sin desinstalar la aplicación.

En un dispositivo Android, el usuario puede utilizar su memoria externa para almacenar sus canciones, sus fotos y sus vídeos. Por lo tanto, esta memoria es susceptible de no estar disponible (lectura/escritura) mientras se usa una aplicación. Por este motivo, es necesario comprobar la disponibilidad del almacenamiento externo antes de realizar cualquier tipo de petición de escritura o lectura. El método **getExternalStorageState** le permite comprobar dicha disponibilidad.

```
final String storageState =
Environment.getExternalStorageState();

if (storageState.equals(Environment.MEDIA_MOUNTED)) {
   //Puede leer y escribir en el almacenamiento externo
} else if
(storageState.equals(Environment.MEDIA_MOUNTED_READ_ONLY)) {
   //Solamente puede leer del almacenamiento externo
} else if (storageState.equals(Environment.MEDIA_REMOVED)) {
   //El almacenamiento externo no está disponible
} else {
   //Resto de casos
}
```

En este ejemplo, puede comprobar el estado del almacenamiento externo entre los diferentes valores disponibles en la clase **Environment**.

A continuación se muestran los diferentes estados posibles:

- **MEDIA_BAD_REMOVAL**: el almacenamiento se ha desconectado sin haber sido desmontado.
- **MEDIA_CHECKING**: el almacenamiento está siendo comprobado.
- **MEDIA_MOUNTED**: el almacenamiento está disponible en modo lectura/escritura.
- **MEDIA_MOUNTED_READ_ONLY**: el almacenamiento está disponible en modo sólo lectura.
- **MEDIA_NOFS**: el almacenamiento está disponible pero utiliza un formato no soportado.
- **MEDIA_REMOVED**: el almacenamiento no está presente.
- **MEDIA_SHARED**: el almacenamiento se usa en USB (conectado al PC).
- **MEDIA_UNMOUNTABLE**: el almacenamiento está presente pero no puede ser montado.
- **MEDIA_UNMOUNTED**: el almacenamiento está disponible pero no está montado.

La escritura en el almacenamiento externo requiere la permission**WRITE_EXTERNAL_STORAGE**.

2. Acceder a los archivos de una aplicación

Puede acceder a los archivos guardados por su aplicación en el almacenamiento externo del dispositivo:

- A partir de la API 8 (Android 2.2), utilice el método **getExternalFileDir(String type)**, que permite obtener una instancia de la clase **File** que apunta al directorio en el que debe guardar sus archivos.
- El método recibe por parámetro una cadena de caracteres que representa el tipo de la carpeta que se abrirá. Este parámetro puede adquirir los siguientes valores:
  - **DIRECTORY_ALARMS**: ubicación de sonidos que pueden usarse como alarma.
  - **DIRECTORY_DCIM**: ubicación de imágenes y fotos.
  - **DIRECTORY_DOWNLOADS**: ubicación de archivos descargados.
  - **DIRECTORY_MOVIES**: ubicación de películas, series y otros vídeos.
  - **DIRECTORY_MUSIC**: ubicación de música.
  - **DIRECTORY_NOTIFICATION**: ubicación de sonidos usados para las notificaciones.
  - **DIRECTORY_PICTURE**: ubicación de imágenes accesibles para el usuario.
  - **DIRECTORY_PODCAST**: ubicación de archivos de audio que representan podcasts.
  - **DIRECTORY_RINGTONES**: ubicación de sonidos utilizados como tonos de llamada.
  - **null**: devuelve la carpeta raíz de su aplicación.

```
File outFile = getExternalFilesDir(Environment.DIRECTORY_DCIM);
```

La nueva versión de Android (Jelly Bean) permite proteger (con una opción) el almacenamiento externo (en modo lectura) mediante la permission **READ_EXTERNAL_STORAGE**.

Estos archivos se eliminarán cuando se desinstale su aplicación.

3. Acceder a archivos compartidos

Los archivos compartidos se guardan en un directorio accesible para el usuario y el resto de aplicaciones. Estos archivos no se eliminarán cuando se desinstale su aplicación, ya que pueden utilizarse, potencialmente, en otras aplicaciones.

Utilice este método solamente para archivos que se compartan con otras aplicaciones.

Estos archivos se encuentran en la raíz del almacenamiento externo y puede acceder a ellos, utilice el método **getExternalStoragePublicDirectory(String type)**. El argumento **type** puede tener los mismos valores que los enumerados en el apartado anterior.

```
File sharedFile =
```

```
Environment.getExternalStoragePublicDirectory(Environment.
DIRECTORY_DCIM);
```

Almacenamiento en base de datos

Cada dispositivo Android incluye una base de datos **SQLite** que puede utilizar en sus aplicaciones para almacenar diferentes tipos de datos.

SQLite es una base de datos relacional Open Source; sólo necesita una pequeña cantidad de memoria en su ejecución.

La sintaxis de **SQLite** se parece a la de SQL y los tipos utilizados son los mismos (INTEGER, TEXT...).

Una base de datos **SQLite** es privada y sólo se puede acceder directamente a ella a través de la aplicación que la ha creado. Se encuentra en una ubicación específica del dispositivo: **data/data/package_de_la_aplicación/databases**

Las operaciones de lectura/escritura no deben realizarse desde el UI thread, sino que hay que externalizarlas en otro **Thread** o en una **AsyncTask**, por ejemplo (véase el capítulo Tratamiento en tareas en segundo plano).

Para la creación de una base de datos, necesitará varios elementos:

- **SQLiteOpenHelper**: se utiliza para facilitar la creación y la gestión de las distintas versiones de una base de datos.
- **SQLiteDatabase**: sirve para exponer los métodos necesarios para la interacción con una base de datos (creación, borrado, actualización…).
- **Cursor**: representa el resultado de una consulta. Este resultado puede contener 0 o más elementos.

El siguiente ejemplo permite crear una base de datos que contiene los distintos capítulos del libro con una descripción de cada capítulo. La tabla contiene tres columnas:

- **Columna id**: la gestiona automáticamente la base de datos.
- **Columna nombre**: especifica el nombre del capítulo.
- **Columna descripción**: contiene una descripción de las secciones abordadas en cada capítulo.

Para comenzar, hay que crear una clase llamada **Capitulo** que represente una instancia de un capítulo.

```
public class Capitulo {

   private int id;
   private String name;
   private String description;

   public Capitulo() {}

   public Capitulo(String name, String descr) {
      this.name = name;
      this.description = descr;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }
```

```
   public void setName(String name) {
      this.name = name;
   }

   public String getDescription() {
      return description;
   }

   public void setDescription(String desc) {
      this.description = desc;
   }

   @Override
   public String toString() {
      StringBuilder sb = new StringBuilder();
      sb.append("Nombre del capítulo = " + name + "\n" +
"Descripción del capítulo =  " + description);
     return sb.toString();
   }

}
```

Esta clase representa un capítulo mediante un identificador, un nombre y una descripción. El método**toString** sirve para mostrar el contenido de un capítulo.

A continuación, hay que crear una clase que extienda a la clase **SQLiteOpenHelper**. Esta clase servirá para crear y gestionar las diferentes versiones de base de datos.

```
public class CapituloBaseSQLite extends SQLiteOpenHelper {

   private static final String TABLA_CAPITULOS = "tabla_capitulos";
   private static final String COL_ID = "ID";
   private static final String COL_NAME = "NAME";
   private static final String COL_DESC = "DESCRIPTION";

   private static final String CREATE_BDD = "CREATE TABLE " +
TABLA_CAPITULOS  + " (" + COL_ID + " INTEGER PRIMARY KEY
AUTOINCREMENT, " + COL_NAME + " TEXT  NOT NULL, " + COL_DESC + "
TEXT NOT NULL);";

   public CapituloBaseSQLite(Context context, String name,
CursorFactory  factory, int version) {
      super (context, name, factory, version);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      db.execSQL(CREATE_BDD);
   }

   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int
 newVersion) {
      //En este método, debe gestionar las actualizaciones de
versión de su base de datos
      db.execSQL("DROP TABLE " + TABLA_CAPITULOS);
      onCreate(db);
   }

}
```

Esta clase dispone de varios atributos:

- El nombre de la tabla (**tabla_capitulos**).
- Los identificadores de las tres columnas que componen la tabla.
- La consulta de creación de la tabla, sin olvidar especificar la columna **que identifica** la clave primaria de la tabla, autoincrementada. Esto permite gestionar fácilmente y de forma automática los identificadores de los distintos registros de la tabla.

La clase dispone de tres métodos:

- **Un constructor**: permite crear un objeto que facilita la creación y la gestión de una base de datos. Recibe cuatro parámetros:
  - **El contexto**: utilizado para crear la base de datos.
  - El nombre del archivo que representa la base de datos.
  - **CursorFactory**: utilizado para sobrecargar la clase que gestiona la creación de Cursors de su base de datos (un cursor permite acceder a los resultados de una consulta en base de datos). Puede pasar el valor null para utilizar la fábrica por defecto.
  - La versión de base de datos.
- **onCreate**: permite crear la base de datos mediante la consulta almacenada previamente, usando el método **execSQL**.
- **onUpgrade**: este método se utiliza cuando se incrementa la versión de la base de datos. Debe gestionar esta actualización de versión así como la compatibilidad respecto a la anterior versión de la tabla.

A continuación, hay que crear la clase que servirá para gestionar los datos (inserción, consulta, borrado...) de la tabla.

```
public class CapituloBBDD {

   private static final intVERSION = 1;
   private static final String NOMBRE_BBDD = "capitulo.db";

   private static final String TABLA_CAPITULOS = "tabla_capitulos";
   private static final String COL_ID = "ID";
   private static final intNUM_COL_ID = 0;
   private static final String COL_NAME = "NAME";
   private static final intNUM_COL_NAME = 1;
   private static final String COL_DESC = "DESCRIPTION";
   private static final intNUM_COL_DESC = 2;

   private SQLiteDatabase bbdd;
   private CapituloBaseSQLite capitulos;
   public CapituloBBDD(Context context) {
        capitulos = new CapituloBaseSQLite(context, NOMBRE_BDD, null, VERSION);
   }

   public void openForWrite() {
      bbdd = capitulos.getWritableDatabase();
   }

   public void openForRead() {
      bbdd = capitulos.getReadableDatabase();
    }

    public void close() {
       bbdd.close();
    }

    public SQLiteDatabase getBbdd() {
```

```
        return bbdd;
    }

    public long insertChapter(Capitulo capitulo) {
        ContentValues content = new ContentValues();
        content.put(COL_NAME, capitulo.getName());
        content.put(COL_DESC, capitulo.getDescription());
        return bbdd.insert(TABLA_CAPITULOS, null, content);
    }

    public int updateChapter(int id, Capitulo capitulo) {
        ContentValues content = new ContentValues();
        content.put(COL_NAME, capitulo.getName());
        content.put(COL_DESC, capitulo.getDescription());
        return bbdd.update(TABLA_CAPITULOS, content, COL_ID + " = " +
id, null);
    }

    public int removeChapter(String name) {
        return bbdd.delete(TABLA_CAPITULOS, COL_NAME + " = " +
name, null);
    }

    public Capitulo getChapter(String name) {
        Cursor c = bbdd.query(TABLA_CAPITULOS, new String[] {
COL_ID,  COL_NAME,COL_DESC },COL_NAME + " LIKE \"" + name + "\"",  null,
null, null,  COL_NAME);
        return cursorToChapter(c);
    }

    public Chapitre cursorToChapter(Cursor c) {
        if (c.getCount() == 0) {
            c.close();
            return null;
        }
        Capitulo chapter = new Capitulo();
          chapter.setId(c.getInt(NUM_COL_ID));
          chapter.setName(c.getString(NUM_COL_NAME));
          chapter.setDescription(c.getString(NUM_COL_DESC));
          c.close();
          return chapter;
    }

    public ArrayList<Capitulo> getAllChapters() {
          Cursor c = bbdd.query(TABLA_CAPITULOS, new String[] {
COL_ID,  COL_NAME,COL_DESC }, null, null, null, null, COL_NAME);
          if (c.getCount() == 0) {
              c.close();
              return null;
          }
          ArrayList<Capitulo> chapterList = new
ArrayList<Capitulo> ();
          while (c.moveToNext()) {
             Capitulo chapter = new Capitulo();
             chapter.setId(c.getInt(NUM_COL_ID));
             chapter.setName(c.getString(NUM_COL_NAME));
             chapter.setDescription(c.getString(NUM_COL_DESC));
             chapterList.add(chapter);
          }
         c.close();
         return chapterList;
    }
}
```

Esta clase tiene los siguientes atributos:

- La versión de la tabla.
- El nombre de la base de datos (**capitulo.db**).
- El nombre de la tabla.
- El nombre y el identificador de cada columna.
- Una variable que representa la base de datos (variable **bbdd**).
- Una variable que representa la clase **OpenHelper** creada anteriormente.

La clase implementa los siguientes métodos:

- **Constructor**: permite inicializar la instancia de la clase **OpenHelper**.
- Dos métodos **Open**: permiten abrir la conexión a la base de datos en modo lectura o escritura.
- **Close**: permite cerrar la conexión a la base de datos.
- **getBBdd**: permite recuperar la instancia de la base de datos.
- **insertChapter(Capitulo capitulo)**: permite insertar un capítulo en una tabla. Para insertar datos en la base de datos hay que utilizar la clase **ContentValues**, que permite albergar una lista de claves/valores que representan el identificador de una columna y el valor que se insertará. A continuación, invoque al método insert con el nombre de la tabla, los valores que se insertarán y un segundo parámetro (null en este caso).
- **updateChapter(int id, Capitulo capitulo)**: permite actualizar un capítulo en la tabla. Para actualizar datos en una base de datos hay que utilizar la clase **ContentValues**, que contiene una lista de claves/valores que representan el identificador de una columna y el valor que se actualizará. A continuación, se debe invocar al método **update** con el nombre de la tabla, los nuevos valores y la condición que identifica el registro que se actualizará como parámetros.
- **removeChapter(String name)**: permite eliminar un capítulo a partir de su nombre. Para ello, se utiliza el método **remove** con el nombre de la tabla y la condición que identifica el registro que se eliminará.
- **getChapter(String name)**: permite obtener un capítulo en función de su nombre y, a continuación, transformar el **Cursor** obtenido en una instancia de la clase Capitulo.
- **cursorToChapter**: permite transformar un **Cursor** en un Capitulo.
- **getAllChapters**: permite obtener todos los capítulos almacenados en una tabla.

Modifique la actividad para insertar datos en la base de datos que se ha creado y, a continuación, mostrarlos en una lista.

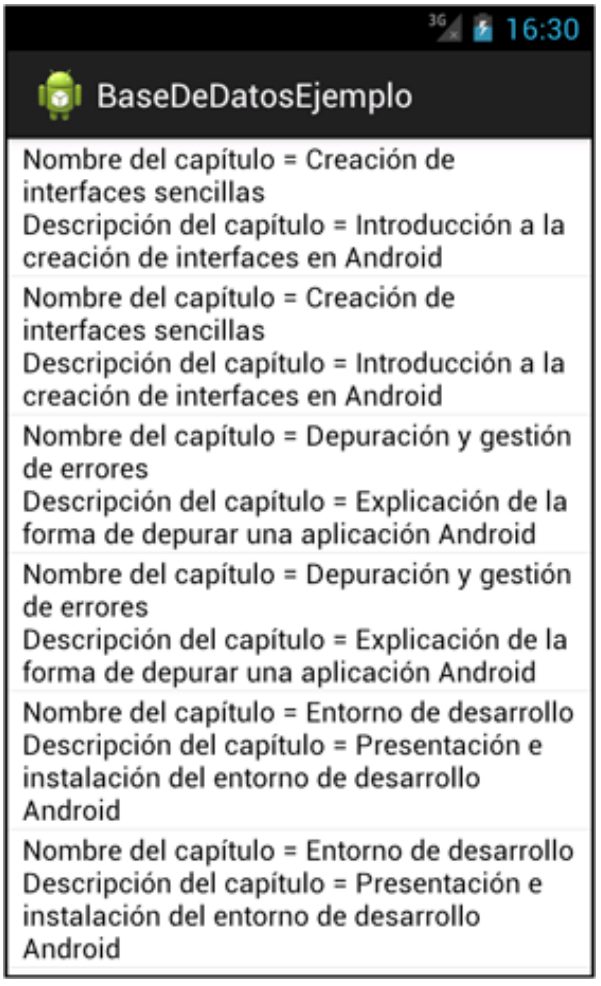
```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);

   ListView list = (ListView) findViewById(R.id.chapterList);

   Capitulo capitulo1 = new Capitulo("La plataforma Android",
"Presentación y cronología de la plataforma Android");
   Capitulo capitulo2 = new Capitulo("Entorno de
desarrollo", "Presentación e instalación del entorno
de desarrollo Android");

   CapituloBBDD capituloBBdd = new CapituloBBDD(this);
   capituloBBdd.openForWrite();
   capituloBBdd.insertChapter(capitulo1);
   capituloBBdd.insertChapter(capitulo2);
```

```
   ArrayList<Capitulo> chapterList =
capituloBBdd.getAllChapters();
   capituloBBdd.close();


   ArrayAdapter<Capitulo> adapter = new
ArrayAdapter<Capitulo>(this,
android.R.layout.simple_list_item_1, chapterList);
   list.setAdapter(adapter);
}
```

La inserción de datos en una base de datos se realiza siguiendo los siguientes pasos:

- Inicialización de la base de datos.
- Apertura de una conexión con la base de datos (en modo escritura).
- Inserción de datos.
- Cierre de la conexión.

Lo que dará:



Puede ver el contenido de la base de datos por línea de comandos y, en particular, con las herramientas **adb** y **sqlite3**.

Para empezar, compruebe que el dispositivo en cuestión está correctamente conectado y que ha sido correctamente detectado por **adb**.

```
adb devices
List of devices attached
016B7DFE0101001A device
```

A continuación, conéctese al dispositivo mediante el siguiente comando:

```
adb shell
```

Ahora, puede acceder a la carpeta que contiene la base de datos creada por la aplicación y enumerar los archivos albergados:

```
# cd /data/data/com.eni.android.database/databases
cd /data/data/com.eni.android.database/databases
# ls
ls
capitulo.db
capitulo.db-journal
```

A continuación, acceda a la base de datos (**capitulo.db**) mediante el comando **sqlite3**, siga las siguientes instrucciones:

```
# sqlite3 capitulo.db
sqlite3 capitulo.db
SQLite version 3.7.4
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

A continuación, es fácil consultar los datos de una tabla.

```
select * from tabla_capitulos;
1|La plataforma Android|Presentación y cronología de la
plataforma Android
2|Entorno de desarrollo|Presentación e instalación del
entorno de desarrollo Android
3|Principios de programación|Presentación de las características
específicas deldesarrollo de Android
```

Para más información, consulte la ayuda (comando **.help**).

La base de datos sólo es accesible en un emulador y no en un dispositivo, salvo si tiene permisos para administrar el dispositivo.

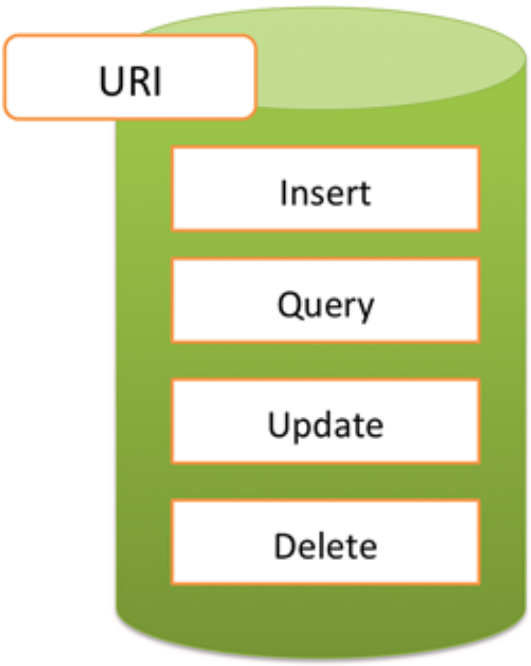
ContentProvider

Un **ContentProvider** (proveedor de contenidos) permite a una aplicación compartir sus datos (almacenados en una base de datos SQLite, en un archivo, en sharedPreferences...) con el resto de aplicaciones.

Además, Android también ofrece varios **ContentProvider** que son accesibles por las aplicaciones (audio, vídeo, imágenes, contactos...).

Un **ContentProvider** se compone:

- De una URI (enlace al proveedor de contenidos).
- De varios métodos que permiten gestionar los datos (**insert**, **query**, **update**, **delete**).



Para acceder a un **ContentProvider**, debe utilizar su URI. Ésta se compone de las siguientes partes:

A: un prefijo estándar que sirve para identificar que el tipo de URI de la solicitud corresponde a un ContentProvider.

B: permite identificar a la autoridad responsable de la gestión del ContentProvider al que deseamos acceder.

C: permite especificar la colección de datos a la que deseamos acceder (tipo de dato). Este segmento es opcional pero necesario si el ContentProvider gestiona varias colecciones de datos.

D: el identificador del dato deseado. Este segmento es opcional y su ausencia significa que se desea obtener todos los datos del segmento.

1. Crear un ContentProvider

Para crear un **ContentProvider**, hay que seguir los siguientes pasos:

- Implementar un sistema para almacenar los datos (base de datos SQLite, archivos, sharedPreferences...).

- Crear una clase que extienda de la clase **ContentProvider**.
- Declarar el **ContentProvider** en el manifiesto de la aplicación.
- Sobrecargar los siguientes métodos:
  - **query**: permite obtener un dato almacenado en un proveedor de contenidos. Este método devuelve un **Cursor**.
  - **insert**: permite insertar un dato en un proveedor de contenidos.
  - **update**: permite actualizar un dato ya existente en un proveedor de contenidos.
  - **delete**: permite eliminar un dato ya existente en un proveedor de contenidos.
  - **getType**: devuelve el tipo **MIME** correspondiente al proveedor de contenidos.
  - **onCreate**: utilice este método para inicializar el proveedor de contenidos.

El ejemplo que se detalla a continuación representa un ContentProvider que permite gestionar una lista de capítulos, cada uno de ellos albergando un identificador, un título y una descripción.

Se utilizará como punto de partida el proyecto creado en el apartado dedicado a las bases de datos (véase el capítulo Persistencia de datos).

Para comenzar, cree una clase que herede de **ContentProvider** y sobrecargue los seis métodos anteriores.

El primer método que se sobrecargará es **onCreate**:

```
@Override
public boolean onCreate() {
   dbHelper = new CapituloBaseSQLite(getContext(),
CapituloBBDD.NOMBRE_BDD, null,CapituloBBDD.VERSION);
   return true
}
```

Incializa el ContentProvider creando una instancia de la clase **CapituloBaseSQLite** (que hereda de**SQLiteOpenHelper** y sirve para conectarse con la base de datos). El método debe devolver "verdadero" si se inicializa con éxito y "falso" en caso contrario.

A continuación, implemente el método **getType**:

```
private final String CONTENT_PROVIDER_MIME =
"vnd.android.cursor.item/vnd.eni.android.database.provider.capitulos";

@Override   public String getType(Uri uri) {
   return CONTENT_PROVIDER_MIME;
}
```

Este método debe devolver el tipo MIME del proveedor de contenidos. El tipo **MIME** permite identificar los datos devueltos y debe comenzar por **vnd.android.cursor.item**/ para consultas de un solo elemento o por **vnd.android.cursor.dir**/ para consultas de múltiples elementos.

La siguiente parte del tipo debe respetar la sintaxis siguiente:

/vnd.<nombre_de_la_autoridad_que_gestiona_el_provider>.<tipodecontenido>

El método **getId** permite obtener el identificador del elemento consultado gracias a la URI que recibe como parámetro.

```
public long getId(Uri contentUri) {
   String lastPathSegment = contentUri.getLastPathSegment();
   if (lastPathSegment != null) {
```

```
        return Long.parseLong(lastPathSegment);
    }
    return -1;
}
```

Para comenzar, es necesario descodificar la última parte de la URI para recuperar la parte que describe el elemento consultado (parte D en el esquema de descripción anterior). Una vez se ha obtenido esta parte, basta con conseguir el entero correspondiente. Si la parte correspondiente al elemento no existe en la URI, el método devuelve -1 para indicar la ausencia de identificador. Esto indica que la consulta afecta a una colección de datos.

El método **insert** permite insertar un elemento en el proveedor de contenidos. Recibe como parámetro la URI del ContentProvider así como los elementos que se insertarán.

```
@Override
public Uri insert(Uri uri, ContentValues values) {
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    try {
     long id = db.insertOrThrow(CapituloBBDD.TABLA_CAPITULOS,
null,values);

    if (id == -1) {
        throw new RuntimeException(String.format(
        "%s : Failed to insert [%s] for unknown reasons.",
        "MyAndroidProvider", values, uri));
    } else {
        return ContentUris.withAppendedId(uri, id);
    }

} finally {
        db.close();
    }
}
```

El primer paso consiste en abrir una conexión en modo escritura con la base de datos del proveedor de contenidos e insertar los elementos en el ContentProvider usando el método **insertOrThrow**.

Este método devuelve un valor. Si este valor es igual a -1, significa que se ha producido un error en la inserción; en caso contrario este valor se corresponde con el identificador en el proveedor de contenidos del elemento insertado. Para finalizar, no olvide cerrar la conexión con la base de datos.

El valor de retorno corresponde a la URI que identifica al elemento insertado.

El método **query** permite obtener el conjunto de elementos alojados en un proveedor de contenidos. El número de elementos obtenidos depende de la URI que se pasa como parámetro.

```
@Override
public Cursor query(Uri uri, String[] projection, String
selection,String[] selectionArgs, String sortOrder) {
    long id = getId(uri);
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    if (id < 0) {
        return db.query(CapituloBBDD.TABLA_CAPITULOS,
        projection, selection, selectionArgs, null,
null,sortOrder);
    } else {
        return db.query(CapituloBBDD.TABLA_CAPITULOS,
        projection, CapituloBBDD.COL_ID + "=" + id, null, null, null,
null);
    }
```

```
}
```

Comience recuperando el identificador correspondiente a la URI que se ha pasado como argumento. Si es menor que 0, significa que la consulta se realiza sobre todo el conjunto de elementos del ContentProvider. En caso contrario, significa que la consulta afecta a un elemento en particular.

Los métodos **update** y **delete** se comportan del mismo modo que el método **query**:

```
@Override
public int update(Uri uri, ContentValues values, String
selection, String[] selectionArgs) {
   long id = getId(uri);
   SQLiteDatabase db = dbHelper.getWritableDatabase();

   try {
      if (id < 0)
      return db.update(CapituloBBDD.TABLA_CAPITULOS, values,
                  selection, selectionArgs);
      else
      return db.update(CapituloBBDD.TABLA_CAPITULOS, values,
            CapituloBBDD.COL_ID + "=" + id, null);
      } finally {
         db.close();
   }
}
```

```
@Override
   public int delete(Uri uri, String selection, String[]
selectionArgs) {
      long id = getId(uri);
      SQLiteDatabase db = dbHelper.getWritableDatabase();
      try {
         if (id < 0)
            return
db.delete(CapituloBBDD.TABLA_CAPITULOS, selection,
                  selectionArgs);
         else
            return
db.delete(CapituloBBDD.TABLA_CAPITULOS,
                  CapituloBBDD.COL_ID + "=" + id, selectionArgs);
      } finally {
         db.close();
      }
   }
```

- Para acabar, cree una actividad que permita insertar y mostrar los elementos disponibles en un proveedor de contenidos.

El método **onCreate** permite inicializar los datos que se usarán en la inserción y en la visualización.

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   insertRecords();
   displayContentProvider();
}
```

El método **insertRecords** permite insertar datos en el proveedor de contenidos.

```
private void insertRecords() {
   ContentValues contact = new ContentValues();
   contact.put(CapituloBBDD.COL_NAME, "Capítulo 1");
   contact.put(CapituloBBDD.COL_DESC,
      "Presentación de la plataforma Android");
   getContentResolver().insert(CapituloContentProvider. contentUri, contact);

   contact.clear();
   contact.put(CapituloBBDD.COL_NAME, "Capítulo 2");
   contact.put(CapituloBBDD.COL_DESC, "Entorno de
desarrollo");
   getContentResolver().insert(CapituloContentProvider. contentUri, contact);

   contact.clear();
   contact.put(CapituloBBDD.COL_NAME, "Capítulo 3");
   contact.put(CapituloBBDD.COL_DESC,
   "Principios de programación en Android");
   getContentResolver().insert(CapituloContentProvider. contentUri, contact);
}
```

El almacenamiento de los datos que se insertarán se realiza en un objeto de tipo **ContentValues**, especificando los valores que se asociarán a las diferentes columnas del proveedor de contenidos. Después, utilice el método **insert**, disponible a través del método **getContentResolver** (que permite obtener una instancia del **ContentResolver** deseado y, por lo tanto, del **ContentProvider**).

La clase **ContentResolver** permite acceder a métodos de interacción con un **ContentProvider**.

El método **displayContentProvider** permite recuperar los datos almacenados en el**ContentProvider**.

```
private void displayContentProvider() {
   String columns[] = new String[] { CapituloBBDD.COL_ID,
      CapituloBBDD.COL_NAME, CapituloBBDD.COL_DESC };
   Uri mCapitulos = CapituloContentProvider.contentUri;
   Cursor cur = getContentResolver().query(mCapitulos, columns,
null, null, null);

   if (cur.moveToFirst()) {
   String name = null;
   do {
   name = cur.getString(cur.getColumnIndex(CapituloBBDD.COL_ID))
+ " "
+ cur.getString(cur.getColumnIndex(CapituloBBDD.COL_NAME))
+ " "
+ cur.getString(cur.getColumnIndex(CapituloBBDD.COL_DESC));

   Toast.makeText(this, name + " ", Toast.LENGTH_LONG).show();
   } while (cur.moveToNext());
}
}
```

El primer paso consiste en declarar la lista de columnas que se desea obtener (no necesita recuperar todas las columnas de un proveedor de contenidos, sino solamente aquellas con datos útiles). A continuación, utilice el método **query** para recuperar el conjunto de datos disponibles en el**ContentProvider**.

Recorriendo el **cursor** obtenido se podrá acceder a los datos correspondientes a cada elemento del proveedor de contenidos. Con ello, se mostrará el contenido del elemento (sin olvidarse de pasar al siguiente elemento).

2. Utilizar un ContentProvider

Android proporciona varios **ContentProviders** que puede utilizar en una aplicación (contactos, medios, etc.).

El siguiente ejemplo permite obtener el conjunto de contactos de un dispositivo (solamente los contactos que tengan un número de teléfono).

```
final ContentResolver cr = getContentResolver();
final Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI,
      null, null, null, ContactsContract.Contacts.DISPLAY_NAME
      + " ASC");
final int nameIndex =
cur.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME);
final int hasPhoneIndex =
cur.getColumnIndex(ContactsContract.Contacts.HAS_PHONE_NUMBER);
final int idIndex =
cur.getColumnIndex(ContactsContract.Contacts._ID);

while (cur.moveToNext()) {
   String name = cur.getString(nameIndex);
   String hasNumber = cur.getString(hasPhoneIndex);
   String id = cur.getString(idIndex);

   if (Integer.parseInt(hasNumber) > 0) {
      Log.e("Contacts Loading", "Contact Name = " + name);
      final Cursor pCur = cr.query(

   ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
   null, ContactsContract.CommonDataKinds.Phone.CONTACT_ID
   + " = ?", new String[] { id }, null);

   final int numberIndex = pCur
   .getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER);

   while (pCur.moveToNext()) {
      String number = pCur.getString(numberIndex);
      Log.e("Contacts Loading", "Contact Number  = " + number);
   }

   pCur.close();
   }

   }

cur.close();
```

El primer paso consiste en obtener una instancia de la clase **ContentResolver** para poder acceder al**ContentProvider** deseado. Después, utilice el método query para obtener el conjunto de datos del**ContentProvider**. Para ello, este método recibe los siguientes cinco parámetros:

- El primer parámetro se corresponde con la URI del ContentProvider de contactos.
- El segundo parámetro permite especificar las columnas que se desean recuperar. Puede utilizar el valor **null** si desea obtener todas las columnas.
- El tercer y cuarto parámetros permiten especificar filtros en las columnas. Utilice el valor **null**si desea que no se aplique ningún filtro sobre el resultado.
- El último parámetro permite especificar una forma de ordenar los resultados obtenidos. En este caso, se desea ordenar los resultados según el nombre del contacto de forma ascendiente.

Ahora, hay que obtener el identificador de las columnas deseadas a partir de su nombre

(método**getColumnIndex**). Este método recibe como parámetro el nombre de la columna deseada. En el ejemplo, se obtienen los identificadores de tres columnas (nombre del contacto, presencia de un número de teléfono e identificador del contacto).

A continuación, hay que recorrer los **Cursor** con los valores recuperados y realizar el siguiente tratamiento para cada uno de ellos:

- Obtener los valores de las tres columnas deseadas mediante el método **getString**. Este método recibe como parámetro el identificador de la columna.
- Compruebe si el contacto en cuestión dispone de un número de teléfono. El valor de la columna **HAS_PHONE_NUMBER** es mayor que 0 si el contacto tiene un número de teléfono.
- Obtenga la lista de números de teléfono para cada contacto mediante el método **query**. Puede observar que se filtran los números de teléfono utilizando el identificador del contacto.
- Para terminar, recorra la lista de números obtenida para mostrarlos.

Tenga precaución de no olvidar cerrar el **Cursor** cuando termine de usarlo, y no olvide declarar la *permission* que permite leer los contactos del dispositivo (**READ_CONTACTS**).

Compartir sus datos con otras aplicaciones

El aspecto social y la compartición son muy importantes en el universo de los Smartphones. Los usuarios desean compartir datos importantes entre aplicaciones, pero también en las distintas redes sociales.

La compartición de datos se realiza a través de intents (intenciones). Este mecanismo permite al usuario utilizar las aplicaciones instaladas en su dispositivo para compartir información.

En el siguiente ejemplo, el clic de un botón de la interfaz comparte un asunto y un mensaje.

```
shareSubject = (EditText) findViewById(R.id.shareSubject);
shareMessage = (EditText) findViewById(R.id.shareMessage);
shareBtn = (Button) findViewById(R.id.shareBtn);

shareBtn.setOnClickListener(new OnClickListener() {
@Override
public void onClick(View v) {
Intent intent=new Intent(android.content.Intent.ACTION_SEND);
intent.setType("text/plain");
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
intent.putExtra(Intent.EXTRA_SUBJECT,
shareSubject.getText().toString());
intent.putExtra(Intent.EXTRA_TEXT,
shareMessage.getText().toString());
startActivity(Intent.createChooser(intent,
getResources().getString(R.string.shareData)));
}
});
```

Lo primero que hay que hacer es crear un intent con la acción **SEND** (enviar). Esta acción permite indicar que el intent sirve para enviar datos a otra aplicación.

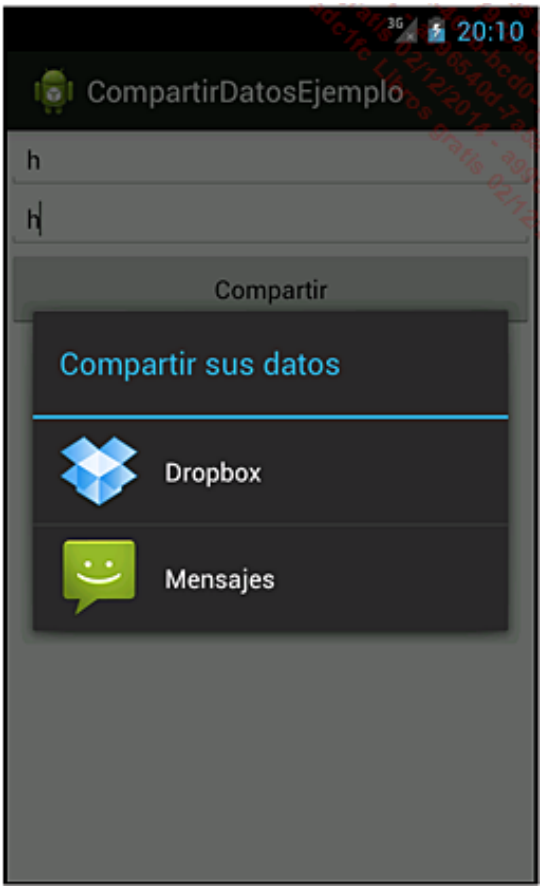
Después, se especifica el tipo de datos que se compartirá (en este caso **text/plain** significa que sólo se compartirá texto). Este tipo se corresponde con el tipo **MIME** de los datos a compartir. Si utiliza el extra **EXTRA_TEXT**, el tipo debe tener como valor **text/plain**.


Si desconoce el tipo que va a utilizar, puede utilizar */*.

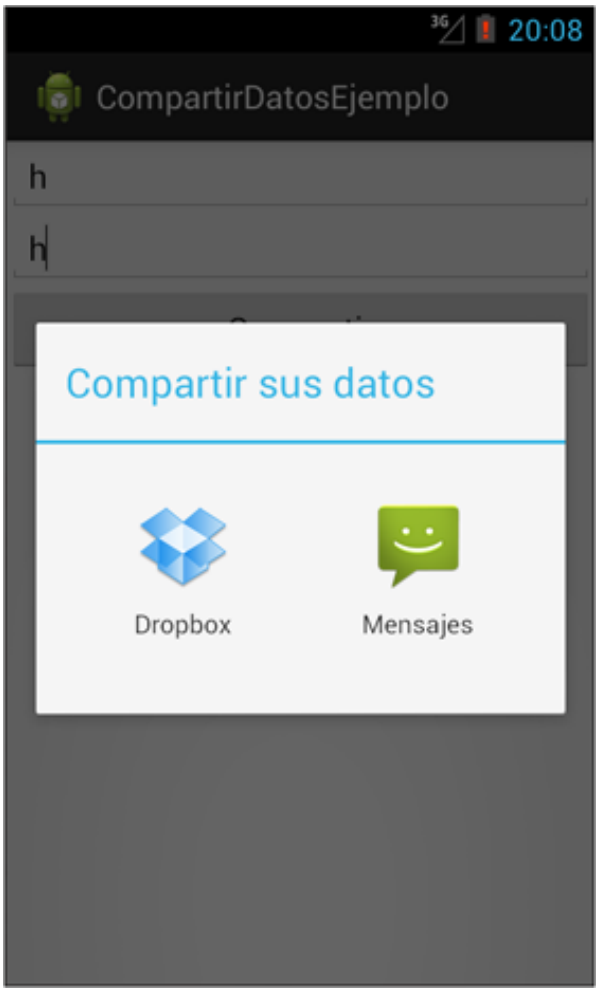
Seguidamente, se especifica el asunto y el texto que se compartirán mediante los extras y los flags**EXTRA_SUBJECT** y **EXTRA_TEXT**.

Para acabar, hay que crear un **Chooser** que permita al usuario elejir con qué aplicación desea compartir los datos.

Lo que generará:



Compartición en Ice Cream Sandwich



Compartición en Jelly Bean

Recibir datos desde otras aplicaciones

Una aplicación también puede recibir datos de otras aplicaciones. Esto es muy práctico sobre todo para aplicaciones con cierto aspecto social (Facebook, Twitter, Google+...) o con matices de compartición (e-mail, SMS...).

Para que una aplicación pueda recibir datos, hay que:

- Modificar el archivo de manifiesto de la aplicación.
- Modificar la actividad que sirve para compartir datos para que sea capaz de recibir el intent que contiene los datos compartidos.

Para comenzar, hay que modificar el archivo de manifiesto de la aplicación para suscribir la aplicación a la recepción de datos. Esta suscripción se puede realizar gracias a los filtros de intenciones (**intent-filter**) y, en particular, gracias a la acción **SEND**.

También debe especificar la categoría **DEFAULT** (indispensable para que la actividad se incluya en la lista de actividades que aceptan la compartición implícita de intents) y el tipo de datos que se comparte.

```
<activity
android:name=".Cap11_RecepcionComparticionDatosEjemploActivity"
            android:label="Mi aplicación" >
    <intent-filter>
       <action android:name="android.intent.action.SEND"/>
       <category android:name="android.intent.category.DEFAULT"/>
       <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

El siguiente paso consiste en modificar la actividad (en concreto el método **onCreate**) especificada en el archivo de manifiesto para recibir los datos compartidos por el usuario.

```
final Intent intent = getIntent();
final String action = intent.getAction();
final String type = intent.getType();
final String text = "text/plain";

if (action.equals(Intent.ACTION_SEND)) {
   if (text.equals(type)) {
       String sharedUrl =
intent.getStringExtra(Intent.EXTRA_TEXT);
        if (sharedUrl != null)
           Toast.makeText(this, sharedUrl,
Toast.LENGTH_LONG).show();
     }
}
```

Para poder obtener datos, hay que:

- Recuperar el intent que se pasa a la actividad (método **getIntent**).
- Recuperar la acción correspondiente al intent.
- Recuperar el tipo de datos compartidos.
- Comprobar que la acción corresponde exactamente a una compartición y que el tipo de datos se corresponde con el tipo de datos esperado.
- Obtener la cadena de caracteres compartida.

Puede recuperar varios datos de forma simultánea mediante la

acción**android.intent.action.SEND_MULTIPLE**.

Lo que dará:

Ice Cream Sandwich

Jelly Bean

Recuperar datos almacenados en línea

1. Conectarse a Internet a través del dispositivo

Una aplicación puede utilizar la conexión a Internet de un dispositivo para acceder a funcionalidades y a datos disponibles en línea.

Hay dos *permissions* básicas que permiten usar esta funcionalidad en el dispositivo:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
```

La primera permite utilizar la conexión a Internet del dispositivo.

La segunda permite comprobar el estado de la red del dispositivo.

Antes de utilizar la conexión a Internet de un dispositivo para realizar una operación (descarga de datos, acceso a un webservice...), hay que comprobar la disponibilidad de la conexión en un instante T para evitar errores debidos a la ausencia de conectividad.

```
ConnectivityManager connectivityManager = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
if (networkInfo != null && networkInfo.isConnected())
{
      // Conexión disponible
} else
{
     // Conexión no disponible
}
```

También puede saber el tipo de conexión activa y, especialmente, si esta conexión corresponde a una conexión Wi-Fi.

```
boolean isWifi = false;
if (networkInfo.getType() == ConnectivityManager.TYPE_WIFI)
   isWifi = true;
```

2. Gestión del cambio de conectividad

Puede hacer que la aplicación se suscriba al cambio de conectividad de un dispositivo. Esta suscripción se puede realizar gracias a la clase **PhoneStateListener** (este cambio de conectividad no incluye Wi-Fi).

```
PhoneStateListener dataState = new PhoneStateListener() {
   @Override
   public void onDataConnectionStateChanged(int state) {
      switch (state) {
         case TelephonyManager.DATA_CONNECTED:
            // Dispositivo conectado
            break;
         case TelephonyManager.DATA_CONNECTING:
            // Dispositivo conectándose
            break;
         case TelephonyManager.DATA_DISCONNECTED:
            // Dispositivo desconectado
            break;
```

```
            case TelephonyManager.DATA_SUSPENDED:
               // Dispositivo conectado pero sin transferencia de datos
               break;
               }
               super.onDataConnectionStateChanged(state);
   }
};

TelephonyManager tm = (TelephonyManager)
getSystemService(TELEPHONY_SERVICE);
tm.listen(dataState,
PhoneStateListener.LISTEN_DATA_CONNECTION_STATE);
```

Este listener permite distinguir cuatro estados que representan la conectividad del dispositivo:

- **DATA_CONNECTED**: el dispositivo está conectado y la red está disponible.
- **DATA_CONNECTING**: el dispositivo se está conectando.
- **DATA_DISCONNECTED**: el dispositivo está desconectado.
- **DATA_SUSPENDED**: el dispositivo está conectado pero la red no está disponible.

Estos datos son indispensables para gestionar el modo sin cobertura en su aplicación y también para evitar una petición a Internet si la conexión no está disponible.

También puede crear un **Broadcast Receiver** (véase el capítulo Tratamiento en tareas en segundo plano - Broadcast Receiver) que le informará del cambio de conectividad del dispositivo.

```
<receiver android:name="MyNetworkStateReceiver">
    <intent-filter>
        <action
android:name="android.net.wifi.p2p.CONNECTION_STATE_CHANGE" />
    </intent-filter>
</receiver>
```

3. Conexión a una dirección remota

Una aplicación Android que tiene las *permissions* necesarias puede conectarse a un dirección remota para realizar una llamada a un webservice, descargar un archivo...

La mayoría de aplicaciones utilizan el protocolo HTTP para establecer conexiones y, en particular, la clase **HttpURLConnection**. Esta clase permite conectarse a un servidor remoto utilizando el protocolo HTTP para enviar y recibir datos.

Esta clase puede utilizarse para realizar streaming y también permite realizar conexiones HTTPS.

Para establecer una conexión con una dirección determinada, hay que seguir los siguientes pasos:

- Crear una instancia de la clase URL mediante la dirección deseada.
- Abrir una conexión (método **openConnection()**) y de este modo obtener una instancia de la clase **HttpURLConnection**.
- Obtener un flujo que permita escribir (**getOutputStream**) o leer (**getInputStream**).
- Realizar el tratamiento deseado.
- Cerrar la conexión mediante el método **disconnect**.

A continuación se muestra un ejemplo que permite conectarse a una dirección para obtener el contenido de un archivo:

```
HttpURLConnection httpUrlConnection = null;
try {
   URL url = new URL(uri);
   httpUrlConnection = (HttpURLConnection) url.openConnection();
   InputStream inStream = new
BufferedInputStream(httpUrlConnection.getInputStream());
   int inChar;
   final StringBuilder readStr = new StringBuilder();
   while ((inChar = inStream.read()) != -1) {
      readStr.append((char) inChar);
   }
   Log.v("HttpConnection", "read String = " + readStr.toString());
   httpUrlConnection.disconnect();
} catch (MalformedURLException me) {
   me.printStackTrace();
} catch (IOException io) {
   io.printStackTrace();
}
```

Una vez se ha logrado establecer la conexión, se realiza la lectura de forma sencilla, utilizando un bucle que va llamando al método **read** (que devuelve -1 cuando se ha llegado al final del archivo). Para finalizar, hay que cerrar la conexión.

Las conexiones **HTTP/HTTPS** siempre deben realizarse en un thread independiente al UI thread de Android (véase el capítulo Tratamiento en tareas en segundo plano), de lo contrario obtendrá la excepción **NetworkOnMainThreadException**.

Si desea, por ejemplo, descargar una imagen almacenada en un servidor remoto, basta con incluir el siguiente código después de obtener el **InputStream**. A continuación, puede asociar esta imagen a un **ImageView** de su aplicación, por ejemplo.

```
Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

4. XML Parsing

Los archivos XML (*eXtensible Markup Language*) se utilizan con mucha frecuencia para obtener los resultados de una petición a un webservice.

Android incluye un parseador XML (**XmlPullParser**, que presenta los elementos que provienen del documento XML en forma de eventos) y ofrece una implementación accesible mediante el método**Xml.newPullParser()**.

A continuación, se muestra un ejemplo de un archivo XML que contiene una lista de capítulos (archivo que se coloca en la carpeta **assets** del proyecto). Cada capítulo contiene:

- Un número.
- Un título.
- Una descripción.

```
<capitulos>
<capitulo id="1" name="La plataforma Android" desc="descripción
capítulo 1" />
<capitulo id="2" name="Entorno de desarrollo"
desc="descripción capítulo 2" />
<capitulo id="3" name="Principios de programación"
desc="descripción capítulo 3" />
<capitulo id="4" name="Mi primera aplicación - HelloAndroid"
desc="descripción capítulo 4" />
```

```
</capitulos>
```

Para poder parsear un archivo XML, hay que seguir los siguientes pasos:

- Obtener una variable de tipo **XmlPullParserFactory** (método **newInstance**), que permite crear el parseador (fábrica de parseadores).
- Crear el **XmlPullParser** mediante el método **newPullParser**.
- Abrir el archivo XML disponible en la carpeta **assets** (**getAssets().open()**) para obtener un flujo en modo lectura sobre el archivo.
- Definir el archivo recuperado como entrada para el parseador XML (método **setInput**).
- Recuperar los eventos descritos en el archivo XML (los eventos corresponden a elementos, atributos…) (método **getEventType**).
- Recorrer todos los eventos hasta alcanzar el evento de fin de documento (**END_DOCUMENT**).

Puede obtener el evento de apertura de un nuevo elemento mediante el evento **START_TAG**.

A continuación, puede obtener el nombre de la etiqueta (**getName**), el número de atributos(**getAttributeCount**) y los valores de los atributos (**getAttributeValue**) de una etiqueta.

El paso al evento siguiente se realiza mediante el método **next**.

No olvide cerrar el flujo cuando finalice la lectura.

```
try {
    XmlPullParserFactory pullParserFactory =
XmlPullParserFactory.newInstance();
    XmlPullParser xmlPullParser =
pullParserFactory.newPullParser();

    InputStream in =
getApplicationContext().getAssets().open("data.xml");
    xmlPullParser.setInput(in, null);

    int eventType = xmlPullParser.getEventType();
    while(eventType != XmlPullParser.END_DOCUMENT) {
       if(eventType == XmlPullParser.START_DOCUMENT) {
            Log.d(TAG, "Inicio del documento");
       } else if(eventType == XmlPullParser.START_TAG) {
            Log.d(TAG, "En el tag = "+
 xmlPullParser.getName());
            int attrsCount = xmlPullParser.getAttributeCount();
            for (int i = 0; i < attrsCount; ++i) {
                Log.d(TAG, "Atributo número " + i + " del tag " +
xmlPullParser.getName() + "es = "+
xmlPullParser.getAttributeValue(i));
            }
       }
       else if(eventType == XmlPullParser.END_TAG) {
            Log.d(TAG, "Fin del tag = "+ xmlPullParser.getName());
       }
       eventType = xmlPullParser.next();
    }
    in.close();
} catch (XmlPullParserException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
```

5. Parsing JSON

JSON (*JavaScript Object Notation*) es el segundo tipo de archivo popular en el uso de webservices. Para parsear este archivo, vamos a utilizar la clase **JSONObject**.

JSON es un formato ligero de intercambio de datos. Es fácil de escribir y de leer, fácilmente analizable y totalmente independiente del lenguaje de programación, lo que lo convierte en un lenguaje de intercambio de datos ideal.

Su estructura se basa en dos elementos:

- Una colección de pares clave/valor.
- Una lista de valores ordenados.

A continuación se muestra el ejemplo de un archivo JSON que contiene algunos capítulos. Cada capítulo contiene un identificador, un nombre y una descripción.

```
{
    "capitulos": {
        "capitulo": [
            {
                "id": "1",
                "name": "La plataforma Android",
                "desc": "descripción capítulo 1"
            },
            {
                "id": "2",
                "name": "Entorno de desarollo",
                "desc": " descripción capítulo 2"
            },
            {
                "id": "3",
                "name": "Principios de programación",
                "desc": " descripción capítulo 3"
            },
            {
                "id": "4",
                "name": "Mi primera aplicación - HelloAndroid",
                "desc": " descripción capítulo 4"
            }
        ]
    }
}
```

Para parsear un archivo JSON hay que seguir los siguientes pasos:

- Obtener el contenido del archivo **JSON** deseado.
- Crear un objeto de tipo **JSONObject** a partir del contenido del archivo JSON.
- Recuperar las distintas etiquetas mediante los métodos (**getJSONObject**: recupera el contenido de una etiqueta y **getJSONArray**: recupera la lista de elementos contenidos en una etiqueta).
- Obtener los diferentes valores contenidos en la lista (etiqueta **capitulo**): identificador, nombre y descripción.

```
public class Cap11_XmlParsingEjemploActivity extends Activity {

   private JSONObject;
   private String jsonStr = "{\"capitulos\": {\"capitulo\":
[{\"id\": \"1\",\"name\": \"La plataforma Android\",\"desc\":
\"descripción capítulo 1\"},{\"id\": \"2\",\"name\":
\"Entorno de desarrollo\",\"desc\": \"descripción
capítulo 2\"},{\"id\": \"3\",\"name\": \"Principios de
programación\",\"desc\": \"descripción capítulo 3\"}]}}";
```

```
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      try {
         parseJsonFile();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   private void parseJsonFile () throws Exception {
      jsonObj = new JSONObject(jString);

      JSONObject capitulos = jsonObj.getJSONObject("capitulos");
      JSONArray capitulo  = capitulos.getJSONArray("capitulo");
      for (int i = 0; i < capitulo.length(); i++) {
         String attributeId =
capitulo.getJSONObject(i).getString("id");
         Log.v("Cap11_XmlParsingEjemploActivity", attributeId);

         String attributeName =
 capitulo.getJSONObject(i).getString(
            "name");
         Log.v("Cap11_XmlParsingEjemploActivity", attributeName);
         String descName =
capitulo.getJSONObject(i).getString("desc");
         Log.v("Cap11_XmlParsingEjemploActivity", descName);
      }
    }
```

Puede utilizar librerías tales como Jacuson para parsear sus archivos XML/JSON.

Introducción

Cuando una aplicación pasa a primer plano, se realiza la ejecución de diferentes tratamientos y tareas en el **UI Thread** (*User Interface Thread*). Este thread es el corazón de su aplicación y se ocupa de toda la gestión de las interfaces y de las interacciones del usuario con la aplicación.

El **UI Thread** ejecutará todo el código presente en un componente de su aplicación (actividad, servicio, proveedor de contenidos, etc.). Por lo tanto, es muy importante que para la experiencia del usuario este thread no tenga ninguna operación pesada y que nunca se bloquee (lo que bloquearía al usuario en su interacción con la aplicación).

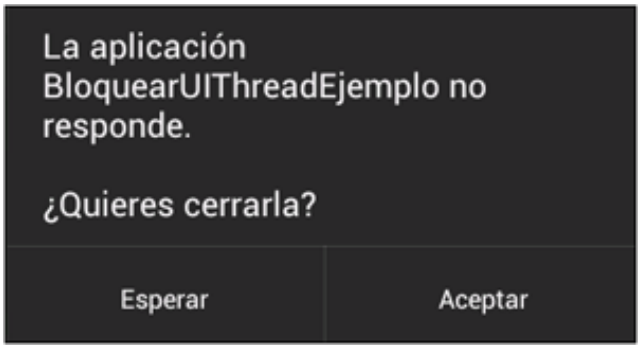
- Para ilustrar este principio, cree un proyecto compuesto por dos botones que simulen el bloqueo del UI Thread.

Para bloquear el UI Thread, asocie un listener al clic del botón 1 que realizará una acción de pausa en el UI Thread (método **sleep**).

```
final Button blockUiThread = (Button)
findViewById(R.id.blockUiThreadBtn);
blockUiThread.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
          try {
              Thread.sleep(15000L);
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
    }
});
```

Cuando el usuario haga clic en el primer botón de la interfaz, ésta quedará inutilizable durante unos segundos (el tiempo que dure el tratamiento).

Si el bloqueo dura demasiado tiempo, se mostrará un cuadro de diálogo preguntando al usuario si desea seguir esperando o detener la aplicación. Este cuadro de diálogo se llama **ANR** (*Application Not Responding* = Aplicación no responde).



Esta ventana da a entender al usuario que su aplicación realiza un tratamiento pesado o costoso en el UI Thread y no en un thread separado. Da la posibilidad de cerrar su aplicación (lo que es perjudicial para la imagen y la popularidad de su aplicación).

Android proporciona varias herramientas que permiten ejecutar los tratamientos pesados de una aplicación en tareas en segundo plano, evitando de este modo el bloqueo del UI Thread.

AsyncTask

Las **AsyncTasks** le permiten ejecutar fácilmente un tratamiento en tareas de segundo plano y recuperar el resultado en la interfaz del usuario sin bloquear el UI Thread.

Para utilizar las **AsyncTasks**, hay que crear una nueva clase que extienda la clase **AsyncTask**.

```
public class myDownloadTask extends AsyncTask<Params, Progress,
Result> {
```

La clase **AsyncTask** se parametriza con tres tipos de datos:

- El tipo de dato que se pasa como parámetro a la clase, en concreto al método**doInBackground**.
- El tipo de datos utilizado para publicar el avance de la tarea en ejecución. Se utiliza en elmétodo **onProgressUpdate** (en una barra de progreso horizontal, por ejemplo).
- El tipo de datos utilizado para publicar el resultado a la interfaz, se transmitirá al método**onPostExecute** a través del método **doInBackground**.

La clase **AsyncTask** le permite sobrecargar los siguientes métodos:

- **onPreExecute**: este método le permite actualizar la interfaz de su aplicación antes de empezar a ejecutar la tarea en segundo plano. Este método se ejecuta en el UI Thread.
- **doInBackground**: este método se ejecuta en un thread separado, lo que le permite ejecutar un tratamiento pesado en una tarea de segundo plano.
- **onProgressUpdate**: este método le permite actualizar el progreso de la tarea en ejecución. Se invoca gracias a la función **publishProgress**.
- **onPostExecute**: este método permite actualizar la interfaz con el resultado obtenido al final del tratamiento ejecutado en el método **doInBackground**.

Para comprender mejor este componente, cree un proyecto Android que tenga un botón. Un clic en este botón provocará la simulación de un tratamiento en una **AsyncTask**.

El progreso del tratamiento se indica mediante una barra de progreso horizontal.

La interfaz se compondrá de dos elementos:

- Un botón.
- Una barra de progreso: tiene un estilo horizontal y está oculta por defecto (atributo **visibility**), el objetivo es mostrarla durante el tratamiento.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="match_parent">

      <Button android:text="@string/launch_async"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:id="@+id/launch_async"
          android:layout_gravity="center_horizontal"
          android:layout_margin="10sp"/>

      <ProgressBar
style="@android:style/Widget.ProgressBar.Horizontal"
        android:id="@+id/progress"
        android:layout_height="wrap_content"
```

```
        android:layout_width="match_parent"
        android:visibility="gone"
        android:layout_margin="10sp"/>

</FrameLayout>
```

Ahora, cree una clase que herede de la clase **AsyncTask**.

```
public class myDownloadTask extends AsyncTask<String, Integer,
String> {
}
```

Puede utilizar **Void** como tipo de datos si no desea utilizar datos en el parámetro (en alguno de los tres tipos definidos durante la creación de una **AsyncTask**).

Seguidamente, implemente el método **onPreExecute** para ocultar el botón y mostrar la barra de progreso.

```
@Override
protected void onPreExecute() {
    super.onPreExecute();
    launchAsync.setVisibility(View.GONE);
    progress.setVisibility(View.VISIBLE);
}
```

A continuación, implemente el método **doInBackground**:

```
@Override
protected String doInBackground(String... params) {
    String uri = params[0];
    String result = "";
    for (int i = 1; i <= 10; ++i) {
       try {
         Thread.sleep(1000L);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
       publishProgress(i * 10);
       result += i;
    }
    return result;
}
```

- Este método recibe como parámetro una tabla sin acotar de cadenas de caracteres (tipo elegido en la sobrecarga de la clase **AsyncTask**).
- Se realiza la simulación de un tratamiento pesado mediante un bucle que recorre del 1 al 10. Con cada iteración se detiene el thread durante 1 segundo y, a continuación, se publica el grado de avance del tratamiento (se aumenta de 10 en 10 en cada iteración).
- Para finalizar, hay que devolver el resultado al método **onPostExecute**.

La llamada al método **publishProgress** provoca la ejecución del método **onProgressUpdate**.

```
@Override
protected void onProgressUpdate(Integer... progress) {
    super.onProgressUpdate(progress);
AsyncTaskActivity.this.progress.setProgress(progress[0]);
}
```

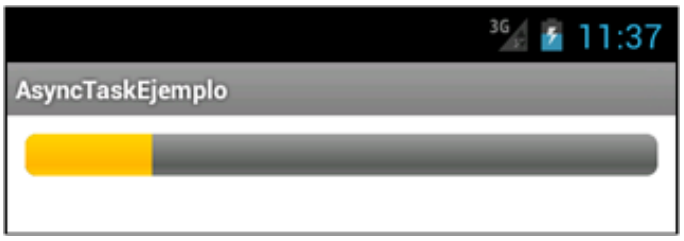
Este método permite definir el grado de progreso actualizando la barra de progreso.

- Ahora, implemente el método **onPostExecute** que, en este ejemplo, sirve para mostrar un Toast indicando la finalización de la tarea en segundo plano así como para ocultar la barra de progreso y mostrar el botón.

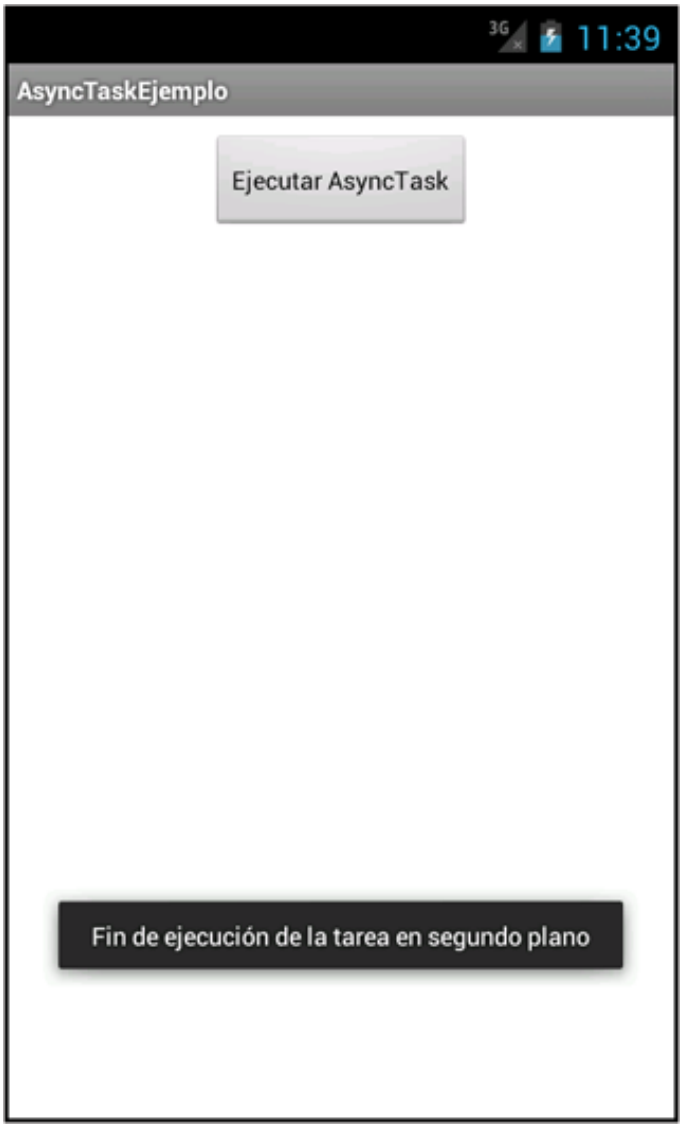
```
@Override
protected void onPostExecute(String result) {
    super.onPostExecute(result);
    Toast.makeText(AsyncTaskActivity.this, "Finalización del
tratamiento en segundo plano", Toast.LENGTH_LONG).show();
    launchAsync.setVisibility(View.VISIBLE);
    progress.setVisibility(View.GONE);
}
```

Con lo que se obtendrá:

Antes de la ejecución del tratamiento



Durante la ejecución del tratamiento



Después de la ejecución del tratamiento

En lo concerniente a las **AsyncTasks**, debe recordar los siguientes puntos:

- Sólo el método **doInBackground** es el que no se ejecuta en el UI Thread.
- Las **AsyncTasks** no persisten si se mata la actividad, tras un cambio de orientación del dispositivo, por ejemplo.
- Su uso es ideal para tratamientos cortos.
- Una **AsyncTask** no puede ejecutarse hasta que la ejecución anterior finalice.
- El tipo **Void** evita el uso de uno de los tipos de datos definidos en la sobrecarga de una**AsyncTask**.
- La ejecución de la **AsyncTask** se realiza mediante el método **execute**:

```
downloadTask = new myDownloadTask();
downloadTask.execute(uri);
```

Thread y Handler

Las **AsyncTasks** utilizan, en su funcionamiento interno, el principio de **Thread**, **Handler** y **Message**.

- Un **Thread** es un contexto de ejecución en el que se ejecuta una serie de instrucciones. En Android, se crea un thread con cada arranque de aplicación (el UI Thread).
- Un **Handler** permite interactuar con un thread proporcionándole instrucciones (**Message**) que se deben ejecutar.
- Un **Message** representa un comando que se debe ejecutar, se envía al thread utilizando el Handler.

En una **AsyncTask**, el método **doInBackground** crea un nuevo thread para ejecutar los tratamientos como tarea en segundo plano. La llamada al método **publishProgress** equivale a un mensaje enviado utilizando un Handler y permite ejecutar el método **onProgressUpdate**.

Para ilustrar estos ejemplos, se creará un proyecto realizando el mismo tratamiento que en el ejemplo anterior (véase la sección AsyncTask) utilizando únicamente Threads, Handlers y Messages.

Comience identificando los distintos mensajes que se deben usar en la aplicación:

- El mensaje que permite mostrar la barra de progreso.
- El mensaje que permite actualizar el avance.
- El mensaje que permite ocultar la barra de progreso.

```
private static final int MESSAGE_PRE_EXECUTE = 1;
private static final int MESSAGE_PROGRESS_UPDATE = 2;
private static final int MESSAGE_POST_EXECUTE = 3;
```

A continuación, implemente los tres métodos correspondientes a los tres mensajes definidos anteriormente.

```
private void downloadOnPreExecute() {
   launchAsync.setVisibility(View.GONE);
   progress.setVisibility(View.VISIBLE);
}

protected void downloadOnProgressUpdate(int progress) {
    AsyncTaskActivity.this.progress.setProgress(progress);
}

protected void downloadOnPostExecute() {
    Toast.makeText(AsyncTaskActivity.this,
        "Fin de la ejecución del tratamiento en segundo plano",
        Toast.LENGTH_LONG).show();
    launchAsync.setVisibility(View.VISIBLE);
    progress.setVisibility(View.GONE);
}
```

Ahora, cree una instancia de la clase **Handler** para gestionar los distintos mensajes enviados al thread.

```
downloadHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        switch (msg.what) {
           case MESSAGE_PRE_EXECUTE:
```

```
                downloadOnPreExecute();
                break;
            case MESSAGE_PROGRESS_UPDATE:
                downloadOnProgressUpdate(msg.arg1);
                break;
            case MESSAGE_POST_EXECUTE:
                downloadOnPostExecute();
                break;
            default:
                break;
        }
    }
};
```

- Esta instancia sobrecarga el método **handleMessage** para obtener los distintos mensajes que se envían al thread y poder ejecutar el método correspondiente.
- El método **handleMessage** recibe como parámetro el mensaje enviado.
- El tipo del mensaje se encuentra en la variable **what**.
- En función del mensaje recibido, ejecute el método correspondiente.
- El **Handler** se ejecuta en el UI Thread.

El último paso consiste en crear la instancia de la clase **Thread**.

```
downloadThread = new Thread(new Runnable() {
    @Override
    public void run() {

        sendPreExecuteMessage();

        for (int i = 1; i <= 10; ++i) {
           try {
              Thread.sleep(1000L);
           } catch (InterruptedException e) {
                e.printStackTrace();
             }

                sendProgressMessage(i);
            }

            sendPostExecuteMessage();
        }

        private void sendPostExecuteMessage() {
            Message postExecuteMsg = new Message();
            postExecuteMsg.what = MESSAGE_POST_EXECUTE;
            downloadHandler.sendMessage(postExecuteMsg);
        }

        private void sendPreExecuteMessage() {
            Message preExecuteMsg = new Message();
            preExecuteMsg.what = MESSAGE_PRE_EXECUTE;
            downloadHandler.sendMessage(preExecuteMsg);
        }

        private void sendProgressMessage(int i) {
            Message progressMsg = new Message();
            progressMsg.what = MESSAGE_PROGRESS_UPDATE;
            progressMsg.arg1 = i * 10;
            downloadHandler.sendMessage(progressMsg);
        }
```

```
});
```

- El thread realiza los mismos tratamientos que la **AsyncTask**.
- Los distintos mensajes se envían en momentos precisos del **Thread**.
- El tipo del mensaje se define mediante el atributo **what**.
- El paso de información puede realizarse mediante el atributo **arg1**.

Puede ejecutar el thread mediante el método **start**.

```
if (!downloadThread.isAlive())
      downloadThread.start();
```

El método **isAlive** permite evitar la ejecución de un thread dos veces, lo que generaría una excepción.

Servicios

Los servicios son componentes Android que no tienen interfaz gráfica. Funcionan de forma invisible para el usuario ejecutando un tratamiento o una tarea. Permiten ejecutar tratamientos bastante largos que no dependen de las interacciones del usuario.

Una de las particularidades de los servicios reside en la prioridad asignada por el sistema Android. Un servicio ejecutando una tarea en segundo plano es prioritario respecto a una actividad ejecutando también una tarea en segundo plano, lo que hace que los servicios estén menos expuestos a la liberación de recursos por el sistema.

Un servicio es (sólo) menos prioritario que una aplicación ejecutándose en primer plano, lo cual reduce la probabilidad de que el sistema mate a sus servicios.

1. Crear y utilizar un servicio

Para declarar un servicio, hay que realizar dos pasos:

- En primer lugar, cree una clase que extienda de la clase **Service**.
- Después, declare el servicio en el archivo de manifiesto de su aplicación.

La nueva clase que sobrecargue a la clase **Service** implementa los siguientes dos métodos:

- **onCreate**: inicialización del servicio y de su entorno.
- **onBind**: permite asociar un servicio a una actividad (véase la sección Asociar un servicio a una actividad).

Con lo que se obtiene:

```
public class MyFirstService extends Service {

   @Override
   public IBinder onBind(Intent intent) {
       return null;
   }

   @Override
   public void onCreate() {
       super.onCreate();
   }

}
```

Sin olvidar añadir la declaración del servicio en el manifiesto de la aplicación.

```
<application android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

    <service android:name=".MyFirstService" />

</application>
```

La ejecución de un servicio (método **startService**) corresponde a la llamada al método**onStartCommand**. Este método sirve para ejecutar el tratamiento que debe realizar su servicio.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
```

```
    return super.onStartCommand(intent, flags, startId);
}
```

Este método recibe tres parámetros:

- El primer parámetro representa una instancia de la clase **Intent**, puede tener valores distintos según el valor de retorno del método **onStartCommand**(**START_STICKY**/**START_NOT_STICKY**/**START_REDELIVER_INTENT**).
- El segundo parámetro permite saber las condiciones de ejecución del servicio (0,**START_FLAG_REDELIVERY**, **START_FLAG_RETRY**).
- El tercer parámetro especifica la acción que se deberá ejecutar.

El método **onStartCommand** devuelve un valor entero que sirve para especificar el comportamiento de su servicio. Este entero puede tener los siguientes valores:

- **START_STICKY**: significa que, si el sistema mata el servicio, automáticamente se reiniciará si hay recursos disponibles. Se llamará automáticamente al método **onStartCommand** (no se conservará el estado del servicio).
- **START_NOT_STICKY**: significa que, si el sistema mata al servicio, no se reiniciará automáticamente aunque haya recursos disponibles.
- **START_REDELIVER_INTENT**: significa que, si el sistema mata al servicio, se reiniciará automáticamente si hay recursos disponibles recibiendo como parámetro el estado (**intent**) anterior del servicio. Esto será así hasta que el servicio llame al método **stopSelf**.

Para **ejecutar** un servicio, puede utilizar el método **startService**:

```
Intent serviceIntent = new Intent(this, MyFirstService.class);
startService(serviceIntent);
```

Si el método **startService** se invoca en el UI Thread, la ejecución del servicio se realizará en el UI Thread. Deberá crear un nuevo **Thread** o una **AsyncTask** antes de comenzar a ejecutar el tratamiento que realizará su servicio. Por ejemplo:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId)
{
    new Thread(new Runnable() {
        @Override
        public void run() {
           doServiceWork();
        }
    }).run();
    return START_STICKY;
}
```

Hay dos modos de detener un servicio:

- Desde el exterior del servicio: un componente Android puede detener un servicio mediante el método **stopService**.

```
Intent serviceIntent = new Intent(this, MyFirstService.class);
stopService(serviceIntent);
```

- Desde el interior del servicio: un servicio puede poner fin a su ejecución mediante el método **stopSelf**.

Puede combinar un servicio con un Broadcast Receiver (**BOOT_COMPLETED**) para iniciar automáticamente el servicio tras el arranque del dispositivo.

2. Asociar un servicio a una actividad

Un servicio puede necesitar actualizar una actividad. Por ejemplo, la reproducción de música puede actualizar el nombre del título en reproducción cuando la aplicación está en primer plano.

Esta asociación se puede realizar gracias al método **onBind** implementado en la sobrecarga de la clase **Service**.

- Debe comenzar creando una clase que herede de **Binder**. Esta clase servirá para asociar una actividad a un servicio.

```
public class MyActivityBinder extends Binder {
   MyFirstService getMyService() {
      return MyFirstService.this;
   }
}
```

- Después, cree una instancia de la clase **MyActivityBinder**.

```
private IBinder myBinder = new MyActivityBinder();
```

- Ahora, modifique el método **onBind** para devolver la instancia que acaba de crear.

```
@Override
public IBinder onBind(Intent intent) {
    return myBinder;
}
```

- Para finalizar, cree una variable de tipo **ServiceConnection** en su actividad. Con ello se podrá gestionar la conexión (**onServiceConnected**) y la desconexión (**onServiceDisconnected**) de la actividad con el servicio.

```
private MyFirstService myService;

private ServiceConnection myServiceConnection = new
ServiceConnection() {

    @Override
    public void onServiceDisconnected(ComponentName name) {
       myService = null;
    }

    @Override
    public void onServiceConnected(ComponentName name, IBinder
service) {
       myService = ((MyFirstService.MyActivityBinder)
service).getMyService();
   }
};
```

- El último paso consiste en asociar y ejecutar el servicio a partir de la actividad.

```
Intent bindIntent = new Intent(this, MyFirstService.class);
bindService(bindIntent, myServiceConnection,
Context.BIND_AUTO_CREATE);
```

Una vez que el servicio se ha asociado a la actividad, todos los métodos y todos los atributos públicos del servicio serán accesibles desde la actividad.

Puede utilizar la clase **IntentService** para crear un servicio que pueda reaccionar a eventos asíncronos.

Broadcast Receiver

El sistema de **Broadcast Receiver** (Receptor de eventos) permite enviar y recibir eventos mediante intents (intenciones).

1. Recibir un evento

Para poder recibir un evento mediante un Broadcast Receiver hay que crear una clase que sobrecargue la clase **BroadcastReceiver**. Esta clase tendrá solamente el método **onReceive**, que se invoca cuando se recibe un evento al que se ha suscrito la clase.

El primer paso consiste en definir el tipo de evento al que se desea suscribir. En el siguiente ejemplo se creará un Broadcast Receiver que permita suscribirse a la recepción de mensajes.

```
<receiver android:name="MySmsReceiver" android:exported="false">
  <intent-filter>
  <action android:name="android.provider.Telephony.SMS_RECEIVED" />
  </intent-filter>
</receiver>
```

- El atributo **exported** permite indicar si otras actividades que no pertenezcan a su aplicación pueden usar el Broadcast Receiver.
- El elemento **action** permite especificar el tipo de evento que desea recibir (**SMS_RECEIVED** = Recepción de mensajes).

Todo ello sin olvidar la *permission* que permite recibir SMS.

```
<uses-permission android:name="android.permission.RECEIVE_SMS" />
```

Puede registrar un Broadcast directamente en el método **onResume** de una actividad, por ejemplo:

```
IntentFilter filter = new IntentFilter();
filter.addAction("android.provider.Telephony.SMS_RECEIVED");
BroadcastReceiver smsReceiver = new BroadcastReceiver() {
@Override
   public void onReceive(Context context, Intent intent) {
   }
};
registerReceiver(smsReceiver, filter);
```

Si registra un Broadcast dinámicamente en el método **onResume** de una actividad, debe anular este registro en el método **onPause**.

```
unregisterReceiver(smsReceiver);
```

Ahora, cree la clase **MySmsReceiver**.

```
public class MySmsReceiver extends BroadcastReceiver {

   @Override
   public void onReceive(Context context, Intent intent) {

   }
}
```

A continuación, implemente el método **onReceive**:

```
private final static String ACTION_RECEIVE_SMS =
"android.provider.Telephony.SMS_RECEIVED";

@Override
public void onReceive(Context context, Intent intent) {

if (intent.getAction().equals(ACTION_RECEIVE_SMS)) {
Bundle bundle = intent.getExtras();
if (bundle != null) {
   Object[] pdus = (Object[]) bundle.get("pdus");

   final SmsMessage[] messageList = new SmsMessage[pdus.length];
   int pos = 0;
   for (Object msgPdu : pdus) {
        messageList[pos] = SmsMessage.createFromPdu((byte[])
msgPdu);
      ++pos;
   }

   for (SmsMessage message : messageList) {
        final String messageBody = message.getMessageBody();
        final String messagePhoneNb = message
   .getDisplayOriginatingAddress();

        Toast.makeText(context, "Emisor: " +
messagePhoneNb, Toast.LENGTH_LONG).show();

        Toast.makeText(context, "Mensaje: " + messageBody,
        Toast.LENGTH_LONG).show();
   }
}
}

}
```

- El primer paso consiste en comprobar que el evento recibido es la recepción de un mensaje.
- A continuación, obtenga el contenido del mensaje (véase el capítulo Telefonía - Gestión de mensajes).
- Para finalizar, muestre el mensaje y el emisor en un **Toast**.

Si un Broadcast Receiver sólo se utiliza en su aplicación (es local a su aplicación), utilice la clase**LocalBroadcast** en vez de la clase **BroadcastReceiver**.

2. Enviar un evento

También puede enviar Broadcasts desde su aplicación gracias al método **sendBroadcast**.

Debe definir la acción correspondiente al Broadcast que quiere propagar.

```
String MY_BROADCAST =
"com.eni.android.broadcast.MY_CUSTOM_BROADCAST";

Intent sendBroadcast = new Intent(MY_BROADCAST);
sendBroadcast(sendBroadcast);
```

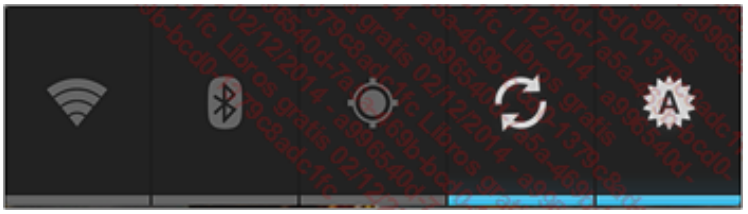
Puede añadir datos en el Broadcast mediante el método **putExtra**.

Widget

1. Presentación

Una de las características específicas de Android es la posibilidad que ofrece a los usuarios de configurar widgets en los escritorios del dispositivo. Los widgets permiten al usuario tener acceso a funcionalidades o a información directamente desde el escritorio, sin tener que iniciar la aplicación en cuestión. Los widgets también pueden ser aplicaciones en sí mismos (el único componente de la aplicación es el widget).

Un widget ocupa un tamaño mínimo predefinido, este tamaño se expresa (para un usuario) en el número de celdas que ocupa.



La versión 4 de Android introduce dos novedades respecto a los widgets:

- La posibilidad de redimensionar a su gusto los widgets.
- La posibilidad de ver una vista previa del widget antes de instalarlo.

El diseño de un widget es distinto al del resto de componentes Android y, por consiguiente, tiene una serie de características específicas:

- Debe definir el tamaño (anchura y altura) mínimo del widget: para conocer el tamaño de un widget en función del número de celdas que debe ocupar, puede utilizar la siguiente función:**Tamaño del widget (altura o anchura) = 70 * Número de celdas ocupadas por el widget - 30**
- Las imágenes utilizadas por un widget deben poder estirarse. Para ello, utilice la herramienta**9-patch** (véase el capítulo Creación de interfaces avanzadas - Buenas prácticas).
- El widget debe ser visible independientemente del fondo de pantalla definido por el usuario.

2. Implementación

Los widgets se basan en los **Broadcast Receivers** y utilizan **RemoteViews** para cargar y actualizar dinámicamente su interfaz.

La creación de un widget se realiza siguiendo los siguientes pasos:

- Crear un archivo XML que describa la interfaz.
- Crear un archivo que describa las características del widget.
- Crear una clase que herede de la clase **AppWidgetProvider** que gestiona el clic y la actualización. Esta clase hereda de **Broadcast Receiver**.
- Declarar los distintos componentes (**AppWidgetProvider**) en el archivo de manifiesto de la aplicación.

Para ilustrar la creación de widgets, se va a crear uno que permite activar/desactivar el modo avión del dispositivo.

Estará compuesto por una imagen y un botón (que permitirá activar/desactivar el modo avión).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/airplane"
        android:layout_gravity="center_horizontal"/>

    <Button
        android:id="@+id/airplane_btn"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/background_grey"
        android:text="@string/airplane_off" />

</LinearLayout>
```

Un widget puede tener los siguientes elementos gráficos:

- **Layouts**: FrameLayout - LinearLayout - RelativeLayout.
- **Componentes**: AnalogClock - Button - Chronometer - ImageButton - ImageView - ProgressBar - TextView - ViewFlipper - ListView - GridView.

Ahora, cree un archivo que represente las propiedades del widget. Este archivo debe ubicarse en la carpeta **xml** (debe crearla en la carpeta **res**).

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider
xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="72dp"
    android:minHeight="72dp"
    android:initialLayout="@layout/widget_layout"
    android:previewImage="@drawable/widget_preview"
    android:resizeMode="horizontal">
</appwidget-provider>
```

Este archivo describe las características específicas del widget:

- **android:minWidth**: anchura mínima ocupada por el widget.
- **android:minHeight**: altura mínima ocupada por el widget.
- **android:initialLayout**: layout que representa la interfaz inicial del widget (hay que crearlo previamente).
- **android:previewImage**: imagen utilizada como presentación del widget antes de la instalación.

Con lo que se obtendrá:

- **android:resizeMode**: descripción de las posibilidades de redimensionamiento ofrecidas por el widget.

A continuación, cree una clase que herede de la clase **AppWidgetProvider**. Esta clase permite que el widget pueda recibir eventos específicos.

Puede sobrecargar los siguientes cinco métodos:

- **onUpdate**: este método se invoca tras cada actualización del widget (atributo**android:updatePeriodMillis** presente en el archivo de configuración del widget). Este método también se invoca cuando un usuario añade el widget en el escritorio del dispositivo.
- **onDelete**: se invoca cuando se elimina el widget del escritorio.
- **onEnabled**: se invoca tras la primera creación del widget.
- **onDisabled**: se invoca cuando se desactiva la última instancia del widget.
- **onReceive**: se invoca cuando el widget recibe un evento.

Comience por el método **onUpdate**:

```
@Override
   public void onUpdate(Context context, AppWidgetManager
appWidgetManager, int[] appWidgetIds) {
```

```
   ntent receiver = new Intent(context, AirPlaneWidget.class);
   receiver.setAction(AIRPLANE_MODE);
   receiver.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS,
appWidgetIds);

   PendingIntent pendingIntent =
PendingIntent.getBroadcast(context, 0,  receiver, 0);
   RemoteViews views = new RemoteViews(context.getPackageName(),
                       R.layout.widget_layout);
   views.setOnClickPendingIntent(R.id.airplane_btn,
pendingIntent);
   appWidgetManager.updateAppWidget(appWidgetIds, views);
}
```

Este método recibe tres parámetros:

- El contexto de la aplicación.
- Un gestor de widget que permite ella ejecución de la actualización mediante el método**updateAppWidget**.
- El identificador de los widgets (puede haber más de uno perteneciente a la misma aplicación) que se actualizarán.

Este método se invoca tras la creación y la actualización del widget. Permite especificar un listener para el clic del botón de la interfaz.

En este método se realizan las siguientes operaciones:

- Crear un receiver que apunte a la clase que sobrecarga el método **onReceive**.
- Crear un **PendingIntent** asociado al receiver declarado anteriormente.
- Cargar la vista que representa la interfaz del widget para posicionar un Pending clic listener en el botón. Cada clic del botón generará un Broadcast Receiver que se obtendrá a través del método **onReceive**.
- Para finalizar, actualice el widget para tener en cuenta las modificaciones aportadas.

Implemente el método **onReceive**.

```
@Override
public void onReceive(Context context, Intent intent) {
   super.onReceive(context, intent);
   if (intent.getAction().equals(AIRPLANE_MODE)) {
       final RemoteViews views = new
RemoteViews(context.getPackageName(),  R.layout.widget_layout);

       boolean isAirplaneEnabled = false;
       if(Settings.System.getInt(context.getContentResolver(),
       Settings.System.AIRPLANE_MODE_ON, 0) != 0) {
          isAirplaneEnabled = true;
       }

       Settings.System.putInt(context.getContentResolver(),
       Settings.System.AIRPLANE_MODE_ON,
       isAirplaneEnabled ? 0 : 1);

       Intent changeMode = new
Intent(Intent.ACTION_AIRPLANE_MODE_CHANGED);
       changeMode.putExtra("state", !isAirplaneEnabled);
       context.sendBroadcast(changeMode);

       if (!isAirplaneEnabled) {
```

```
          views.setInt(R.id.airplane_btn, "setText",
R.string.airplane_on);
       } else {
          views.setInt(R.id.airplane_btn, "setText",
                   R.string.airplane_off);
         }

       AppWidgetManager appWidgetManager = AppWidgetManager
                .getInstance(context);
       appWidgetManager.updateAppWidget(new
ComponentName(context, AirPlaneWidget.class), views);
    }
}
```

Cada llamada al método **onReceive** se corresponde con un clic del usuario en el botón del widget. Está compuesto por los siguientes pasos:

- Comprobar que el evento se corresponde con el evento desencadenado a raíz del clic en el widget.
- Cargar la vista que representa el widget para poder actualizarla (**RemoteView**).
- Comprobar el estado del modo avión del dispositivo (si está activo o no) mediante el método**getInt**, presente en la clase **System** (subclase de **Settings**). El método **getInt** recibe como parámetro el tipo de propiedad solicitada (**AIRPLANE_MODE_ON**).
- Modificar el valor del modo avión del sistema. Si el modo avión estaba activo hay que desactivarlo, y a la inversa. Para ello, utilice los siguientes dos métodos:
  - Utilice el método **putInt** del mismo modo que el método **getInt**.
    - Envíe un Broadcast Receiver para advertir al sistema del cambio de estado del modo avión.
    - La acción de este **broadcast** será: Intent.ACTION_AIRPLANE_MODE_CHANGED
    - Especifique el nuevo valor del estado para el modo avión mediante el extra**state**.
  - Utilice el método **sendBroadcast** para desencadenar el broadcast y advertir al sistema este cambio.
- Actualizar el texto del botón de la interfaz en función del estado del modo avión.
  - Para ello, utilice el método **setInt** que le permite llamar al método **setText** pasando como parámetro el texto a escribir. Este método está disponible en la clase **Button** y es accesible mediante la instancia de la clase **RemoteView**.
- Actualizar el widget y aplicar las modificaciones. Para ello, hay que recuperar una instancia de la clase **AppWidgetManager** para llamar al método **updateAppWidget**, que permite actualizar el widget.

No olvide modificar el archivo de manifiesto de la aplicación añadiendo:

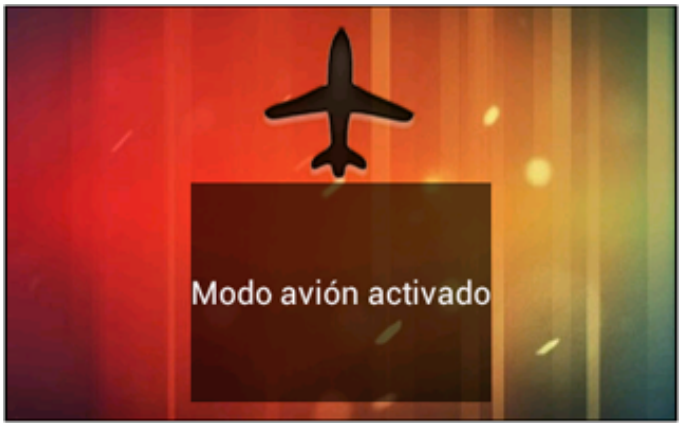
- La *permission* necesaria para modificar los parámetros del dispositivo.
- La declaración del receiver, que tendrá:
  - Una acción de actualización del widget (**APPWIDGET_UPDATE**).
  - Así como un enlace al archivo de configuración (elemento **meta-data**).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.eni.android.widget"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-sdk android:minSdkVersion="15" />
    <uses-permission
android:name="android.permission.WRITE_SETTINGS"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <receiver
            android:name=".AirPlaneWidget"
            android:label="Widget modo avion" >
            <intent-filter>
                <action
android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>

            <meta-data
                android:name="android.appwidget.provider"
                android:resource="@xml/widget_properties" />
        </receiver>

        </application>

</manifest>
```

Lo que generará:

Estado inicial del widget.



Estado del widget después de la activación del modo avión.

3. Jelly Bean

La nueva versión de Android permite:

- Utilizar los **GridLayout** para construir widgets.
- Actualizar los datos (extras) utilizados por un widget (**updateAppWidgetOptions**).

Alarmas

1. Presentación

Una alarma es un intent con un tiempo de disparo o un intervalo predefinidos. Puede utilizarlos para desencadenar eventos periódicos en su aplicación sin que ésta se inicie ni esté en ejecución.

Por ejemplo, en una aplicación de twitter, utilizar una alarma permite actualizar automáticamente los tweets a intervalos regulares sin necesidad de arrancar la aplicación. Esto permite al usuario acceder en todo momento a una lista de tweets actualizada.


Una alarma vuelve a su valor inicial cuando se reinicia el dispositivo.

2. Implementación

La clase **AlarmManager** permite crear y gestionar alarmas en una aplicación. Puede recuperar una instancia de la clase AlarmManager mediante el método **getSystemService**.

```
AlarmManager alarmManager = (AlarmManager)
getSystemService(Context.ALARM_SERVICE);
```

Para crear una alarma, debe utilizar el método **set(int type, long time, PendingIntent intent)**, que recibe tres argumentos:

- El tipo de alarma deseado, que puede ser uno de los siguientes tipos:
  - **ELAPSED_REALTIME**: esta alarma no activa el dispositivo si éste está en espera. Cuenta el tiempo desde que se ha encendido el dispositivo.
  - **ELAPSED_REALTIME_WAKEUP**: esta alarma activa el dispositivo si éste está en espera. Cuenta el tiempo transcurrido desde que se ha encendido el dispositivo.
  - **RTC**: mismo funcionamiento que **ELAPSED_REALTIME** excepto que el tiempo se corresponde con la hora actual.
  - **RTC_WAKEUP**: mismo funcionamiento que **ELAPSED_REALTIME_WAKEUP** excepto que el tiempo se corresponde con la hora actual.
- El tiempo (en milisegundos) de espera hasta que se dispare la alarma.
- La acción que se ejecutará cuando se dispare la alarma.

```
public void onClick(View v) {
   AlarmManager alarmManager = (AlarmManager)
getSystemService(Context.ALARM_SERVICE);
   String myAction = "MY_ACTION";
   Intent intentToLaunch = new Intent(myAction);
   PendingIntent pending =
PendingIntent.getBroadcast(MyCustomAlarm.this,
0, intentToLaunch,  0);
   alarmManager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, 1000L,
pending);
```

Cuando la alarma finaliza la cuenta atrás, se dispara el evento específico.

Para anular una alarma, utilice el método **cancel**.

```
alarmManager.cancel(pending);
```

Este método permite configurar alarmas que sólo se dispararán una vez. También puede crear alarmas que se disparan periódicamente mediante los siguientes dos métodos:

- **setRepeating(int type, long time, long interval, PendingIntent pending)**: permite especificar una alarma que se dispare según un intervalo específico.
- **setInexactRepeating(int type, long time, long interval, Pending- Intent pending)**: permite especificar una alarma que se dispare en un intervalo aproximado.


Tenga especial cuidado con el consumo de batería si utiliza alarmas periódicas.

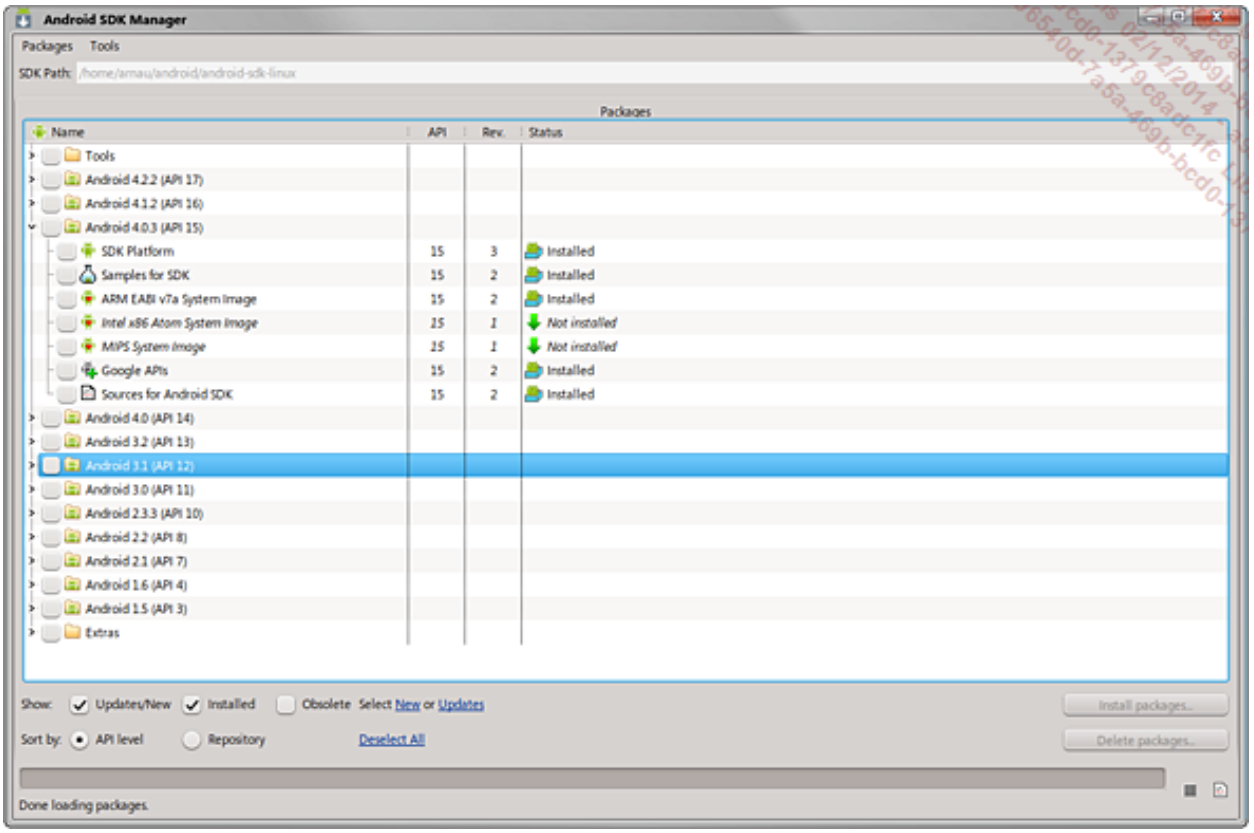
Requisitos

Este capítulo aborda un tema particularmente importante y específico del mundo de los smartphones y de la movilidad en general. Se trata de la geolocalización y la integración de un componente Google Maps en una aplicación Android.

1. Instalación de las APIs de Google

Si desea utilizar los mapas de Google en su aplicación, en primer lugar, hay que instalar la versión del SDK (Kit de desarrollo) 4.0.3 que integra las APIs de Google.

- Para ello, inicie el **SDK Manager** mediante el pequeño icono disponible en Eclipse (véase el capítulo El entorno de desarrollo - Eclipse).



- Compruebe que se haya instalado correctamente la versión 4.0.3 (API Google); en caso contrario, instálela. Para ello, seleccione la versión indicada y, a continuación, haga clic en el botón **Install Packages** (**Instalar paquetes**).

Si realiza pruebas en el emulador, este último también debe ejecutar la versión 4.0.3 de la API de Google para poder mostrar los mapas de su aplicación.

2. Obtención de su clave de Google Maps

Uno de los componentes de las APIs de Google es el **MapView**. Este mapa permite integrar fácilmente un Google Map en una aplicación Android y le brinda el acceso a todos los datos disponibles en Google Maps. Esto explica por qué tiene que obtener una clave y aceptar las condiciones de uso de Google.

Obtener una clave es sencillo y gratuito y se realiza en dos pasos:

- Obtener una firma MD5 (*Message-Digest Algorithm*).

- Obtener una clave de la API. A continuación, debe informar la clave de la API cada vez que utilice un **MapView** en su aplicación.

Antes de generar su firma MD5, hay unos puntos que se deben tener en cuenta acerca de los MapViews y de la clave de la API:

- Para mostrar un Google Map, debe obtener una clave.
- Cada clave se asocia con un certificado de depuración único (véase el capítulo El entorno de desarrollo - SDK Android) o con un certificado de generación de su apk.

a. Generación de su firma MD5

Para generar su firma MD5, debe usar el ejecutable **Keytool**. Viene incluido en la instalación de su entorno Java y se encuentra en la carpeta **bin** de la misma.

A continuación, basta con encontrar la carpeta de su **JRE** (*Java Runtime Environment*) y, en su interior, la carpeta **bin**. Por ejemplo, para Windows 7, se encuentra en la siguiente ubicación:***C:\Program Files\Java\jre6\bin***.

A continuación, ejecute el comando **Keytool** con los siguientes comandos:

```
Keytool -list -alias androiddebugkey
-keystore <rutaASuDebugKeystore>.keystore
-storepass suStorepass  -keypass suKeypass
```

- **list**: sirve para imprimir la firma MD5 de su certificado.
- **keystore<nombreDelKeystore>.keystore**: el nombre de su almacén de claves.
- **storepass<contraseña>**: la contraseña de su almacén de claves.
- **alias<nombreAlias>**: el alias de su contraseña que servirá para generar la firma MD5.
- **keypass<contraseña>**: la contraseña correspondiente a la clave elegida anteriormente.

Por defecto, el storepass y el keypass son "android" (valores por defecto para el**debug.keystore**).

El archivo **debug.keystore** se puede encontrar en las siguientes rutas:

- **Windows 7**: C:\Users\nombreUsuario\.android\debug.keystore
- **Windows Vista**: C:\Users\nombreUsuario\AppData\Local\Android\debug.keystore
- **Windows XP**: C:\Documents and Settings\nombreUsuario\Android\debug.keystore
- **Mac, Linux**: ~/.android/debug.keystore

Otra solución es obtener la ruta mediante Eclipse: haga clic en **Window - Preferences - Android - Build**.

En el intérprete de comandos, sitúese en su directorio Java y, a continuación, ejecute el comando **keytool**.

Si utiliza un archivo de depuración distinto al archivo por defecto (debug.keystore), debe utilizarlo para generar su firma MD5 así como el storepass y el keypass correspondientes.

A continuación se muestra un ejemplo del resultado de la ejecución del comando:

```
[arnau@tulkass .android]$ keytool -list -alias androiddebugkey -keystore debug.keystore -storepass android -keypass [redacted]
androiddebugkey, 03-abr-2013, PrivateKeyEntry,
Huella Digital de Certificado (SHA1): 35:45:CF:CD:F8:14:9D:8E:26:D6:99:46:97:78:21:57:9B:A6:04:BD
```

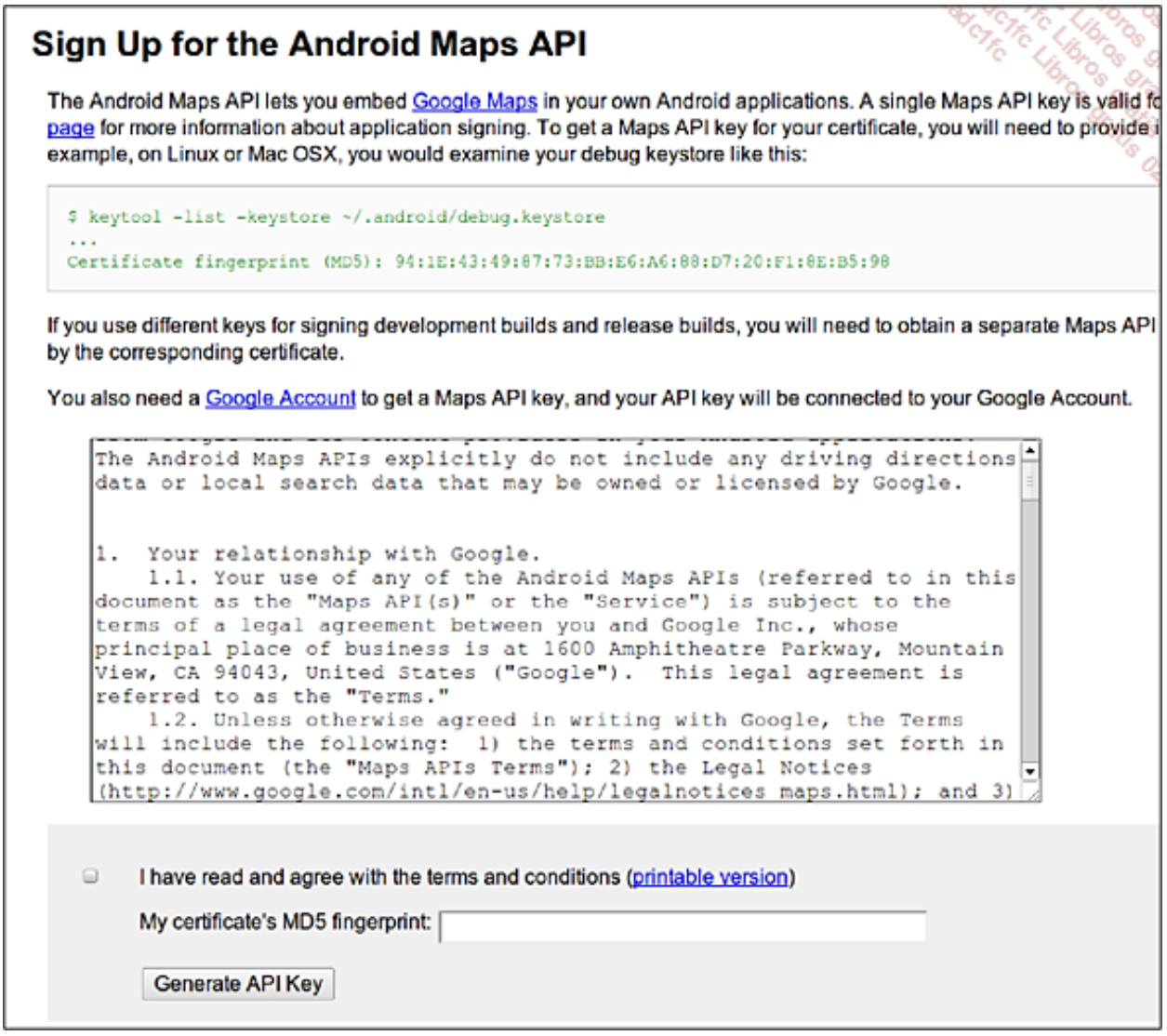
Si sólo obtiene la firma **SHA1**, basta con añadir la opción `-v` en su comando para obtener todas las firmas posibles.

Si aparece el siguiente error: "**Keytool: Comando desconocido**", basta con que se desplace al directorio en el que se encuentra keytool para ejecutar el comando.

b. Obtención de su clave

Con su firma **MD5**, podrá generar su clave.

- Conéctese a: http://code.google.com/android/maps-api-signup.html



Siga el siguiente procedimiento:

- Conéctese con su cuenta de Google, si dispone de una. En caso contrario, deberá crearla.
- Lea los términos del contrato.
- Pegue su firma MD5 en el campo apropiado.
- Haga clic en **Generate API Key** (generar la clave de la API).

El servidor se ocupará de asociar su firma MD5 con su cuenta de desarrollador y le proporcionará

una clave.

La pantalla de resultado le mostrará su clave, su firma MD5 e incluso el código Android que deberá copiar/pegar para crear su Google Map.

Conserve cuidadosamente su clave para evitar tener que generar una nueva cada vez que la necesite.

Integración de un Google Map

1. Creación de la vista Google Map

El proceso de integración de un Google Map en una aplicación Android se puede dividir en varios pasos.

- Para comenzar, indique en el archivo de manifiesto (véase el capítulo Principios de programación - Manifiesto) el uso de las APIs de Google. Para ello, agregue la siguiente línea en la etiqueta **Application** de su manifiesto.

```
<uses-library android:name="com.google.android.maps" />
```

Esta línea permite indicar que su aplicación utiliza las APIs de Google, lo que hará que su aplicación no se pueda instalar en dispositivos que no tengan las APIs de Google Maps.

Agregue también la *permission* para acceder a Internet (véase el capítulo Principios de programación - Permissions (permisos)):

```
<uses-permission android:name="android.permission.INTERNET"/>
```

La *permission* de Internet sirve para mostrar y descargar el mapa.

Con todo ello, el archivo de manifiesto quedará del siguiente modo:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.eni.android.map"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="15" />
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <uses-library android:name="com.google.android.maps" />

        <activity
            android:name=".Ch13_GoogleMapExempleActivity"
            android:label="@string/app_name" >
            <intent-filter>
              <action android:name="android.intent.action.MAIN" />
               <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

- Ahora, modifique el archivo **main.xml**, que se encuentra en la carpeta **res/layout**, para integrarle un **MapView**.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <com.google.android.maps.MapView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:apiKey="@string/mapKey"
        android:id="@+id/map"
    />

</LinearLayout>
```

Observe que el **MapView** no es un elemento nativo de Android y, para utilizarlo, se requiere la librería que se ha integrado anteriormente.

Este **MapView** contiene los siguientes atributos:

- **android:id**: representa el identificador del mapa (véase el capítulo Creación de interfaces sencillas - Principios). La elección de la interfaz es totalmente libre, en este caso se ha elegido el identificador "**map**".
- **android:layout_width** y **android:layout_height**: representan respectivamente la anchura y la altura que ocupará el mapa (véase el capítulo Creación de interfaces sencillas - Principios).
- **android:apiKey**: representa la clave de la API de Google Maps que ha obtenido en el sitio web. Puede almacenarla directamente en el archivo **strings.xml** (véase el capítulo Creación de interfaces sencillas - Recursos).

El siguiente paso consiste en modificar la actividad principal de la aplicación para que herede de laclase **MapActivity** en lugar de la clase **Activity**.

```
public class GoogleMapApiSampleActivity extends MapActivity {

   @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
   }

   @Override
   protectedbooleanisRouteDisplayed() {
      return false;
   }
}
```

Después de haber heredado de la clase **MapActivity**, debe sobrecargar el método**isRouteDisplayed**. Este método debe devolver verdadero (true) si su aplicación muestra rutas o itinerarios.

A continuación se muestra el resultado correspondiente a este ejemplo:

2. Añadir opciones al Google Map

El mapa que se obtiene es básico. No se puede interactuar con el usuario. Las distintas interacciones deben especificarse o implementarse. El objetivo es hacer que el mapa sea más agradable y fácil de usar.

a. Interacción con el usuario

Seguramente se habrá dado cuenta de que el Google Map que ha creado no reacciona a los clics del ratón y no puede desplazarse ni arrastrarlo.

Para resolver este problema, hay que añadir la siguiente línea en la declaración del mapa en el archivo **main.xml**:

```
android:clickable="true"
```

b. Añadir botones de zoom

Hay que permitir que el usuario pueda hacer zoom sin que para ello tenga que hacer doble clic en la pantalla. La solución es añadir los botones que permiten ampliar/reducir.

Para ello, hay que:

- Obtener la instancia del **mapView** mediante su identificador (véase el capítulo Creación de interfaces sencillas - Principios).
- Activar la visualización de los botones de zoom.

- Declare una variable de tipo **MapView** en el archivo que implementa la clase **MapActivity**.

```
private MapView mapView;
```

- Inicialice la instancia de la variable en el método **onCreate** mediante el método **findViewById**.

```
mapView = (MapView) findViewById(R.id.map);
```

Una vez que se ha recuperado la instancia, puede activar los botones de zoom.

```
mapView.setBuiltInZoomControls(true);
```

Si hace clic en el mapa, verá aparecer los botones de zoom:

c. Definir el nivel de zoom

El nivel de zoom del mapa puede parecerle un poco pequeño. En ese caso, puede modificarlo en la inicialización del mapa.

- Obtenga una instancia de la clase **MapController** e invoque al método **setZoom**.

La clase **MapController** le permite gestionar todas las funcionalidades asociadas al zoom y a la inserción de marcadores en el mapa.

- Agregue una variable de tipo **MapController** a la clase.

```
private MapController mapController;
```

- Ahora, inicialice esta instancia mediante el método **getController**. Este método se encuentra disponible a través de la variable de tipo **MapView**.

```
mapController = mapView.getController();
```

- Para finalizar, llame al método **setZoom**, pasándole como parámetro el valor del zoom. Este valor puede estar comprendido entre 1 (zoom mínimo) y 21 (zoom máximo).

```
// Inicialización del valor por defecto del zoom
mapController.setZoom(17);
```

Lo que generará:

d. Visualización en modo satélite

También puede cambiar el tipo del mapa y, de este modo, tener una imagen de satélite en vez de la vista tradicional.

En el ejemplo, la modificación del tipo de mapa mostrado se asociará al uso de los botones de volumen del dispositivo. Para ello, invoque al método **setSatellite** de la clase **MapView**.

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
   if (keyCode == KeyEvent.KEYCODE_VOLUME_UP) {
```

```
            mapView.setSatellite(true);
            return true;
        } else if (keyCode == KeyEvent.KEYCODE_VOLUME_DOWN) {
            mapView.setSatellite(false);
            return true;
        }
        return super.onKeyDown(keyCode, event);
    }
```

Con lo que se obtendrá:

e. Gestión del doble clic

Implementar el zoom con un doble clic en el mapa es un poco más complicado, pero muy útil en la interacción con el mapa. Para empezar, deberá crear una instancia personalizada de la clase**MapView**.

- Para ello, cree una nueva clase que extienda la clase **MapView**.

```
public class MyGoogleMap extends MapView {
    public MyGoogleMap(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

En esta clase, debe implementar un constructor que sirva simplemente para invocar al constructor de la clase **MapView**.

- A continuación, sobrecargue el método **onInterceptTouchEvent** para interceptar el doble clic en el mapa y, de este modo, implementar el siguiente comportamiento.

```
private long lastTouchTime = -1;

@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
   if (ev.getAction() == MotionEvent.ACTION_DOWN) {
       long thisTime = System.currentTimeMillis();
       if (thisTime - lastTouchTime < 250) {
       this.getController().zoomInFixing((int) ev.getX(),
                   (int) ev.getY());
            lastTouchTime = -1;
       } else {
             lastTouchTime = thisTime;
       }
       }
    return super.onInterceptTouchEvent(ev);
}
```

El comportamiento deseado permitirá al usuario hacer zoom automáticamente en el mapa en la ubicación en donde se ha realizado el doble clic.

El primer problema consiste en diferenciar dos clics separados de un doble clic. Para ello, hay que especificar un umbral de tiempo a partir del cual un segundo clic dejará de estar asociado al primero (en este ejemplo, 250 milisegundos). La variable **lastTouchTime** permite conocer el tiempo del último clic para comprobar si la diferencia es menor a 250 milisegundos.

Si el usuario hace doble clic en un punto determinado de la pantalla, invoque al método**zoomInFixing** para hacer zoom en ese mismo punto.

Para poder recuperar el punto correspondiente al clic, hay que utilizar el evento que se pasa como parámetro al método **onInterceptTouchEvent** y obtener las coordenadas a través de los métodos**getX** y **getY**.

- A continuación, reemplace todas las ocurrencias de la clase **MapView** por la clase**MyGoogleMap**.

Lo que dará para el archivo que representa la interfaz principal de la aplicación:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <com.eni.android.map.MyGoogleMap
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:apiKey="@string/mapKey"
         android:id="@+id/map"
         android:clickable="true"/>

</LinearLayout>
```

Puede observar que el mapa usado ya no es el de Google sino el que acabamos de implementar.

- Realice los mismos cambios en la actividad que representa el mapa:

```
public class Ch13_GoogleMapEjemploActivity extends MapActivity {
```

```
    private MyGoogleMap mapView;

    private MapController mapController;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mapView = (MyGoogleMap) findViewById(R.id.map);

        // Añadir botones de zoom
        mapView.setBuiltInZoomControls(true);

        // Inicialización del controlador del mapa
        mapController = mapView.getController();

        // Inicialización del valor por defecto del zoom
        mapController.setZoom(17);

    }

     @Override
     protected boolean isRouteDisplayed() {
         return false;
    }

}
```

Ahora su mapa gestiona el doble clic. Puede utilizar la clase **MapView** personalizada para adaptar, todavía más, el comportamiento del mapa integrado en su aplicación.

Localización

Localizar un usuario es una funcionalidad muy importante en el mundo de la movilidad. Permite, por ejemplo, mostrar contenido específico a la posición del usuario.

Hay dos tipos de localización:

- **Localización exacta**: utilizando el GPS del dispositivo.
- **Localización aproximada**: utilizando elementos que permiten localizar de forma aproximada al usuario (Wi-Fi, 3G, etc.).

A las distintas herramientas que permiten localizar a un usuario (GPS, Wi-Fi, 3G...) se les llama**provider** (proveedor).

La clase que permite gestionar la localización de un usuario es la clase **LocationListener**. Permite recibir notificaciones de posicionamiento del **LocationManager**.

Para añadir esta funcionalidad a una aplicación, basta con que la actividad que representa su mapa implemente la interfaz **LocationListener**.

Esta implementación implica la definición de los siguientes métodos:

- **onLocationChanged(Location location)**: este método se invoca tras actualizar la posición del usuario.
- **onProviderDisabled(String provider)**: se invoca cuando se desactiva un provider. El nombre del provider se pasa como parámetro.
- **onProviderEnabled(String provider)**: se invoca cuando se activa un provider.
- **onStatusChanged(String provider, int status, Bundle extras)**: se invoca cuando el estado de un provider ha cambiado. Existen tres tipos de estado:
  - **OUT_OF_SERVICE**: el provider está inutilizable.
  - **TEMPORARILY_UNAVAILABLE**: el provider está temporalmente no disponible.
  - **AVAILABLE**: el provider está disponible.

Comience por el método **onLocationChanged**:

```
@Override
public void onLocationChanged(Location location) {
   lat = location.getLatitude();
   lng = location.getLongitude();

   p = new GeoPoint((int) (lat * 1E6), (int) (lng * 1E6));
   mapController.animateTo(p);
   mapController.setCenter(p);
}
```

El método **onLocationChanged** proporciona a través del parámetro la nueva localización del usuario, lo que nos permite obtener la latitud y la longitud. Después, se crea una instancia de la clase**GeoPoint**. Esta clase representa un punto en el mapa.

Para finalizar, se desplaza el mapa y su centro a este nuevo punto que representa la posición del usuario.

La implementación de los métodos **onProviderDisabled** y **onProvider-Enabled** permite, simplemente, recibir notificaciones de la activación y la desactivación de un provider. Esta implementación es útil para suscribirse o cancelar la suscripción a las actualizaciones de localización proporcionadas por un provider.

El último método permite detectar el cambio de estado de un provider. Por ejemplo, el paso del GPS del

```
@Override
public void onProviderDisabled(String provider) {
   if (LocationManager.GPS_PROVIDER.equals(provider)) {
       Toast.makeText(this, "GPS provider desactivado",
Toast.LENGTH_SHORT).show();
   } else if (LocationManager.NETWORK_PROVIDER.equals(provider))
{
     Toast.makeText(this, "Network provider desactivado ",
Toast.LENGTH_SHORT).show();
   } else {
      Toast.makeText(this, "otro provider desactivado = " +
provider, Toast.LENGTH_SHORT).show();
   }
}

@Override public void onProviderEnabled(String provider) {
   if (LocationManager.GPS_PROVIDER.equals(provider)) {
       Toast.makeText(this, "GPS provider activado",
Toast.LENGTH_SHORT).show();
   } else if (LocationManager.NETWORK_PROVIDER.equals(provider)) {
       Toast.makeText(this, "Network provider activado ",
Toast.LENGTH_SHORT).show();
   } else {
       Toast.makeText(this, "otro provider activado = " + provider,
Toast.LENGTH_SHORT).show();
   }
}
```

modo búsqueda de señal al modo activación.

```
@Override
   public void onStatusChanged(String provider, int status,
Bundle extras) {
      if (status == LocationProvider.AVAILABLE) {
          Toast.makeText(this, "El provider " + provider +
" está actualmente disponible", Toast.LENGTH_SHORT).show();
      }  else if (status == LocationProvider.OUT_OF_SERVICE) {
          Toast.makeText(this, "El provider " + provider +
" está inutilizable",  Toast.LENGTH_SHORT).show();
      } else if (status ==
LocationProvider.TEMPORARILY_UNAVAILABLE) {
          Toast.makeText(this, "El provider " + provider +
" está temporalmente no disponible", Toast.LENGTH_SHORT).show();
      }
    }
```

La implementación del **LocationListener** no es suficiente, hay que hacer que la instancia del**LocationManager** se suscriba a la actualización de la posición del usuario a través de los providers deseados.

```
locationManager = (LocationManager)
getSystemService(LOCATION_SERVICE);

locationManager.requestLocationUpdates(LocationManager.
GPS_PROVIDER,  10000, 0,  this);
locationManager.requestLocationUpdates(
LocationManager.NETWORK_PROVIDER, 10000, 0, this);
```

La inicialización del **LocationManager** se realiza mediante el método **getSystemService**, tal y como se vio para las notificaciones.

A continuación, se invoca al método **requestLocationUpdates** con los distintos providers. Este método tiene cuatro parámetros:

- El nombre del provider.
- El tiempo mínimo entre dos notificaciones (en milisegundos).
- La distancia mínima entre dos notificaciones (en metros).
- La instancia del **LocationListener**.

Para finalizar, necesita dos *permissions* para poder localizar al usuario.

```
<!-- Localización aproximada: WIFI / 3G... -->
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
<!-- Localización exacta: GPS... -->
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
```

También puede solicitar una única actualización de la posición del usuario mediante el método**requestSingleUpdate**.

Obtener la posición inmediatamente al inicio

Si debe obtener una posición cuando se inicie su aplicación sin tener que esperar a la actualización de la posición o la inicialización de los providers, puede utilizar el método **getLastKnownLocation(String provider)**.

Este método devuelve la última posición conocida por el provider que se pasa como parámetro. Esta posición puede estar obsoleta, por ejemplo, si el usuario se ha movido mientras el teléfono estaba apagado.

El método **getLastKnownLocation** devuelve la posición conocida o **null** si no hay ninguna posición conocida o el provider está inactivo.

Posición del usuario

Ahora su aplicación reacciona correctamente al cambio de posición del usuario pero no existe ningún indicador que permita al usuario ver claramente su posición en el mapa.

Existe una clase en Android que permite, de forma muy sencilla, indicar a un usuario su posición en el mapa. Se trata de la clase **MyLocationOverlay**.

Todos los marcadores que ponga en un mapa se considerarán **overlays**.

- Declare una variable de tipo **MyLocationOverlay**.

```
    private MyLocationOverlay myLocation;
```

- Inicialice la variable declarada anteriormente mediante el constructor de la clase**MyLocationOverlay**. Recibe como parámetro el contexto y la instancia del **MapView**.

```
myLocation = new MyLocationOverlay(this, mapView);
```

- Active la visualización de la posición y agregue el overlay de localización a la lista de overlays que muestra el mapa.

```
myLocation.enableMyLocation();
mapView.getOverlays().add(myLocation);
```

Lo que generará el siguiente resultado:


18:37
GoogleMapsEjemplo
Castellana
CIEN LLAVES
Calle Pedro Muñoz Seca
Iglesia de San José
Plaza Cibeles
Calle de Alcalá
Banco de España
Ayuntamiento de Madrid
Calle Alfonso XI
Instituto Estudio Bursátiles
Paseo del Prado
Calle de Montalbán
Dirección General Marina Mercante
Oficina de Información
Museo Naval
Calle los Madrazo
Calle Juan de Mena
Calle Ruiz de Alarcón
Plaza Lealtad
Calle de Zorrilla
Calle Antonio Maura
Hostal Coruña
Palacio de Villahermosa
Hotel Ritz
Calle Muñoz Núñez

Colocar un marcador en el mapa

Indicar la posición del usuario en un mapa consiste en poner un marcador especial en el mapa, específico a la posición del usuario. Pero también puede añadir marcadores que sirvan, por ejemplo, para indicar puntos de interés en un mapa.

El primer paso para añadir marcadores personalizados en un mapa consiste en crear una clase que represente su lista de marcadores. Para ello, hay que utilizar la clase **ItemizedOverlay** que permite crear y gestionar marcadores más fácilmente.

Para representar su marcador, cree una clase que herede de la clase **ItemizedOverlay**.

```
public class ListaMarcadores extends ItemizedOverlay<OverlayItem> {
```

Esta herencia debe tiparse para especificar la clase que representa su marcador. En nuestro ejemplo, hay que utilizar la clase por defecto.

A continuación, debe sobrecargar los siguientes métodos:

- El constructor de la clase.
- El método **OverlayItem createItem(int i)**.
- El método **int size()**.

Para comenzar, es importante disponer de una lista para gestionar los distintos marcadores de un mapa así como el contexto de la aplicación.

```
private ArrayList<OverlayItem> overlayList;
private Context context;
```

El constructor sirve para inicializar estos valores y especificar la imagen utilizada por sus marcadores.

```
public ListaMarcadores(Drawable drawable, Context context) {
   super(boundCenter(drawable));
   overlayList = new ArrayList<OverlayItem>();
   this.context = context;
}
```

El método **boundCenter** permite centrar la imagen que representa a un marcador en relación a sus coordenadas.

Ahora, cree un método que permita añadir un ítem a la lista definida en los atributos de la clase. Este método debe añadir el ítem que se pasa como parámetro a la lista de marcadores.

La llamada al método **populate** notifica al mapa que se ha añadido un nuevo marcador. Internamente invoca al método **createItem** y ejecuta los procedimientos de dibujo de nuevos marcadores.

```
public void addItemToOverlayList(OverlayItem item) {
   overlayList.add(item);
   populate();
}
```

El método **createItem** devuelve el elemento ubicado en una posición determinada de la lista de marcadores.

```
@Override
protected OverlayItem createItem(int pos) {
   return overlayList.get(pos);
}
```

El método **size** devuelve el número de marcadores presentes en el mapa, que se corresponde con el tamaño de la lista de marcadores.

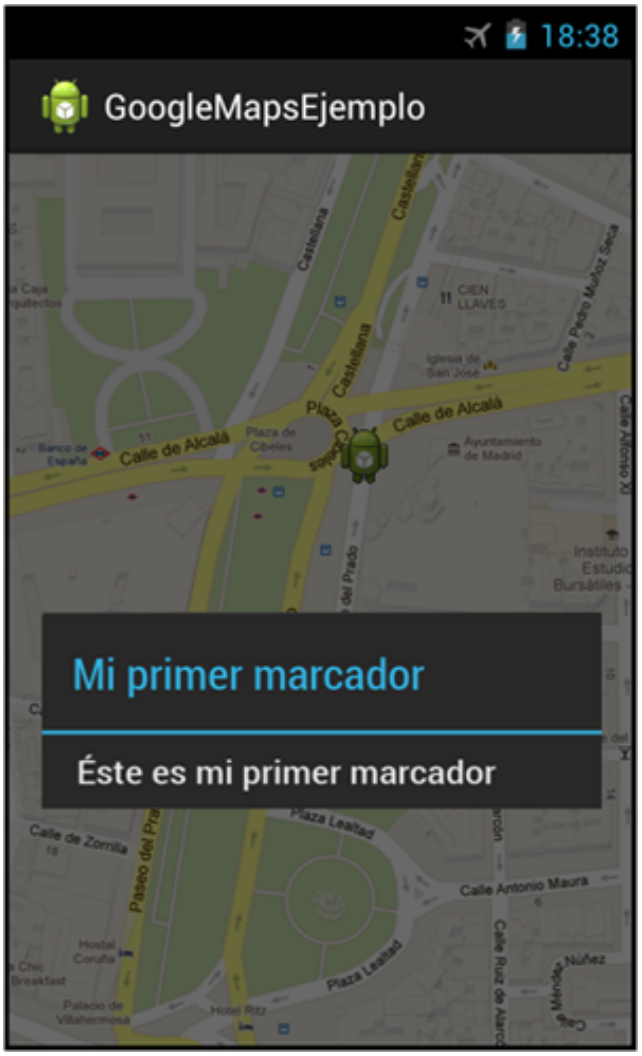
```
@Override
public int size() {
    return overlayList.size();
}
```

Lo que generará:

La gestión del clic se realiza sobrecargando el método **onTap(int index)**. Este método sirve para crear un cuadro de diálogo con la información del marcador seleccionado.

```
@Override
protected boolean onTap(int index) {
   OverlayItem item = overlayList.get(index);
   AlertDialog.Builder dialog = new AlertDialog.Builder(context);
   dialog.setTitle(item.getTitle());
   dialog.setMessage(item.getSnippet());
   dialog.show();
   return true;
}
```

Los métodos **getTitle** y **getSnippet** permiten obtener el título y la descripción del ítem seleccionado. Lo que generará:


18:38
GoogleMapsEjemplo
Mi primer marcador
Éste es mi primer marcador
Calle de Alcalá
Plaza de Cibeles
Castellana
Calle de Alcalá
Ayuntamiento de Madrid
Plaza Lealtad
Calle Antonio Maura
Hotel Ritz
Calle de Zorrilla
Paseo del Prado

Conversión posición/dirección

Gracias a la API de Google Maps, también puede convertir una dirección postal en una posición (latitud/longitud) y viceversa. Esto le permite, por ejemplo, encontrar una posición a partir de una dirección introducida por el usuario o encontrar la dirección en la que se encuentra un usuario a partir de su posición.

- Para ello, utilice la clase **Geocoder**, que permite convertir las coordenadas en direcciones y las direcciones en coordenadas a través de varios métodos (**getFromLocation**,**getFromLocationName**).

El siguiente ejemplo permite mostrar la dirección correspondiente a las coordenadas de las pulsaciones del usuario (método **onTouchEvent**).

```
@Override
public boolean onTouchEvent(MotionEvent event, MapView mapView) {
   if (event.getAction() == MotionEvent.ACTION_UP) {
   GeoPoint clicPosition = mapView.getProjection().fromPixels(
        (int) event.getX(), (int) event.getY());

   Geocoder geoCoder = new Geocoder(context,
Locale.getDefault());
   try {
       List<Address> addr = geoCoder.getFromLocation(
          clicPosition.getLatitudeE6() / 1E6,
          clicPosition.getLongitudeE6() / 1E6, 1);

       String clicAddr = "";
       if (addr.size() > 0) {
           for (int i = 0; i <
addr.get(0).getMaxAddressLineIndex(); i++)
      clicAddr += addr.get(0).getAddressLine(i) + "\n";
      }

   Toast.makeText(context, clicAddr, Toast.LENGTH_SHORT).show();
   } catch (IOException e) {
      e.printStackTrace();
   }
   return true;
} else
   return false;
}
```

El clic de un usuario en un punto del mapa se corresponde con el evento **ACTION_UP**. Con el evento, se obtienen las coordenadas del mapa correspondientes al clic mediante el método**getProjection().fromPixels()**.

Después, se crea una instancia de la clase **Geocoder**. Su inicialización requiere dos parámetros: el contexto de la aplicación y el locale (idioma) del dispositivo.

Puede recuperar el locale del dispositivo mediante el método **getDefault**.

Para finalizar, hay que convertir las coordenadas correspondientes al clic en la lista de direcciones que se recorren para mostrarlas en un **Toast**. Lo que generará:


18:38
GoogleMapsEjemplo
Paseo del Prado
28014, Madrid
Calle de Alcalá
Plaza de Cibeles
Ayuntamiento de Madrid
Calle de Montalbán
Calle Antonio Maura
Plaza Lealtad
Hotel Ritz
Palacio de Villahermosa
Hostal Coruña
Calle de Zorrilla
Calle los Madrazo
Banco de España
Iglesia de San José
CIEN LLAVES
Instituto Estudios Bursátiles
Calle Alfonso XI
Calle Ruiz de Alarcón

Principios

Telefonía (**Telephony**) representa la API de Android que gestiona la voz y los datos (llamadas, SMS, datos, etc.).

En los inicios de Android, todos los dispositivos que tenían esta API -y, por lo tanto, las aplicaciones que utilizaban esta API- no sufrían el problema de fragmentación. Pero desde la llegada de los dispositivos Android equipados con Wi-Fi (como las tablets), el uso de la API por parte de las aplicaciones depende del dispositivo.

La utilización de esta API en una aplicación requiere implementar una gestión para los dispositivos que no la tienen.

Si su aplicación sólo puede usarse en un dispositivo que disponga de esta API, debe especificarlo en su archivo de manifiesto. Esta declaración permitirá que Google Play no ofrezca su aplicación para descargar en dispositivos que no tengan esta API.

A continuación se muestra la declaración que debe añadir en su archivo de manifiesto para especificar este requerimiento:

```
<uses-feature android:name="android.hardware.telephony"
        android:required="true"/>
```

Si el uso de la telefonía es importante en su aplicación pero no imprescindible y simplemente desea desactivar algunas funcionalidades, puede comprobar dinámicamente en su aplicación la presencia o la ausencia de estas funcionalidades mediante el método **hasSystemFeature**.

```
PackageManager packageManager = getPackageManager();
boolean telephonySupported =
packageManager.hasSystemFeature(PackageManager.FEATURE_TELEPHONY);
```

Gestión de llamadas

1. Realizar una llamada

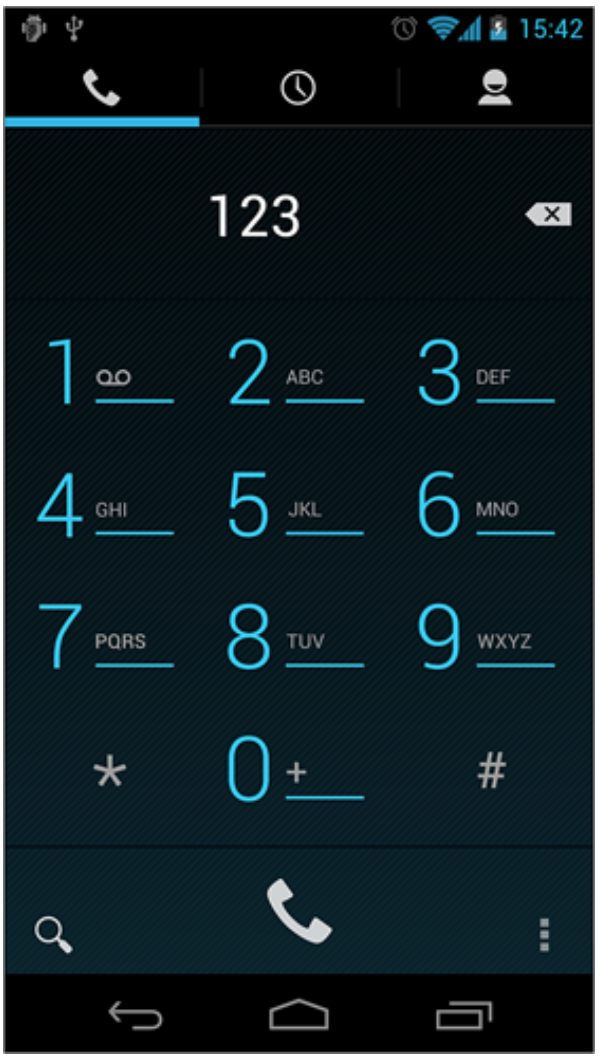
Si el usuario debe poder realizar una llamada desde su aplicación, puede implementar esta funcionalidad simplemente con el uso de un intent que tenga:

- Una acción adecuada (**ACTION_DIAL** o **ACTION_CALL**) para mostrar, respectivamente, el gestor de llamadas o realizar una llamada directamente.
- Un dato que represente el número de teléfono prefijado por **tel:**.

```
Intent call = new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:123"));
startActivity(call);
```

El uso de la acción ACTION_CALL requiere la *permission* CALL_PHONE.

Con lo que obtendrá:



2. Obtener información sobre las propiedades del teléfono

Puede obtener varios datos sobre las propiedades de un dispositivo:

- Tipo del teléfono (CDMA, GSM, SIP, NONE).
- Código IMEI del teléfono.
- Número de teléfono, si está disponible.

Para ello hay que utilizar la clase **TelephonyManager**, que le brindará el acceso a una gran variedad de datos relacionados con la telefonía del dispositivo.

```
TelephonyManager tm = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
// 0 = PHONE_TYPE_NONE, 1 = PHONE_TYPE_GSM, 2 = PHONE_TYPE_CDMA,
3 =  PHONE_TYPE_SIP
int phoneType = tm.getPhoneType();
String imeiCode = tm.getDeviceId();
String phoneNumber = tm.getLine1Number();
// 0 = DATA_DISCONNECTED, 1 = DATA_CONNECTING, 2 =
DATA_CONNECTED, 3 =  DATA_SUSPENDED
int dataState = tm.getDataState();
String operator = tm.getSimOperator();
```

No olvide la *permission*:

```
<uses-permission
android:name="android.permission.READ_PHONE_STATE"/>
```

3. Gestionar los dispositivos entrantes

En algunas aplicaciones, necesitará ser notificado de la llegada de una llamada al dispositivo. Esto le permitirá, por ejemplo, guardar los cambios realizados por el usuario o el progreso en un juego.

Para ello, puede utilizar la clase **PhoneStateListener** y sobrecargar el método **onCallStateChanged**.

Este método le notifica del cambio de estado de las llamadas y, de este modo, reacciona ante uno de los tres casos posibles (inactivo, comunicando, llegada de una llamada).

```
PhoneStateListener phoneStateListener = new PhoneStateListener()
{
    @Override
    public void onCallStateChanged(int state, String
incomingNumber) {
      super.onCallStateChanged(state, incomingNumber);
      switch (state) {
         case TelephonyManager.CALL_STATE_IDLE:
            // Inactivo
            break;
         case TelephonyManager.CALL_STATE_OFFHOOK:
            // Comunicando
            break;
         case TelephonyManager.CALL_STATE_RINGING:
            // Teléfono suena
            break;
         }
    }
};
```

Este listener sólo se ejecuta si se está ejecutando su aplicación.

La lectura de estados de llamada requiere la *permission* **READ_PHONE_STATE**.

```
<uses-permission
android:name="android.permission.READ_PHONE_STATE"/>
```

También puede utilizar un **Broadcast Receiver** para supervisar el estado de las llamadas en el dispositivo.

```
<receiver android:name="MyPhoneStateReceiver">
    <intent-filter>
      <action android:name="android.intent.action.PHONE_STATE"/>
    </intent-filter>
</receiver>
```

Gestión de mensajes

1. Envío de SMS

Una aplicación Android puede enviar mensajes. Para ello, puede usar dos métodos distintos:

- Utilizando un intent invocando la aplicación de mensajes.
- Utilizando la clase **SMSManager**.

El primer método consiste, simplemente, en enviar un intent implícito a la aplicación que gestiona el envío de SMS especificando el número del destinatario y el cuerpo del mensaje.

```
Intent sendSms = new Intent(Intent.ACTION_SENDTO,
Uri.parse("sms:123456"));
sendSms.putExtra("sms_body", "Mi primer SMS");
startActivity(sendSms);
```

Este método no necesita ninguna *permission* ya que no es su aplicación la que enviará el mensaje, sino la aplicación por defecto (especificada por el usuario).

También puede enviar mensajes directamente desde su aplicación mediante la clase **SMSManager**.

Para comenzar, este envío requiere una *permission* (**SEND_SMS**).

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

Hay que obtener una instancia de la clase **SMSManager**. Esta clase es un singleton y, por lo tanto, la instanciación de la clase se realiza con el método **getDefault**.

A continuación, hay que definir el destinatario y el cuerpo del mensaje y, para terminar, utilizar el método **sendTextMessage** para enviar el mensaje.

```
SmsManager smsManager = SmsManager.getDefault();
String receiver = "123456";
String body = "Mi primer SMS";
smsManager.sendTextMessage(receiver, null, body, null, null);
```

El método **sendTextMessage** recibe los siguientes tres parámetros:

- Una cadena de caracteres que representa el destinatario del mensaje.
- Una cadena de caracteres que representa el servicio de envío de mensajes que se utilizará. Puede pasar **null** como valor para utilizar el servicio de envío por defecto.
- Una cadena de caracteres que representa el cuerpo del mensaje.
- Un **PendingIntent** que se ejecutará si el envío del mensaje finaliza (con éxito o con error). Puede pasar **null** para ignorar el evento de envío de mensaje.
- Un **PendingIntent** que se ejecutará tras la recepción del mensaje por parte del destinatario (ya sea con éxito o no). Puede pasar **null** para ignorar el evento de recepción del mensaje.

El tamaño de un mensaje está limitado a 160 caracteres. Si su mensaje supera este tamaño, habrá que dividirlo en múltiples partes mediante el método **divideMessage** disponible a través de la instancia de la clase **SMSManager**.

```
SmsManager smsManager = SmsManager.getDefault();
String body = ".....................";
String receiver = "..............";
```

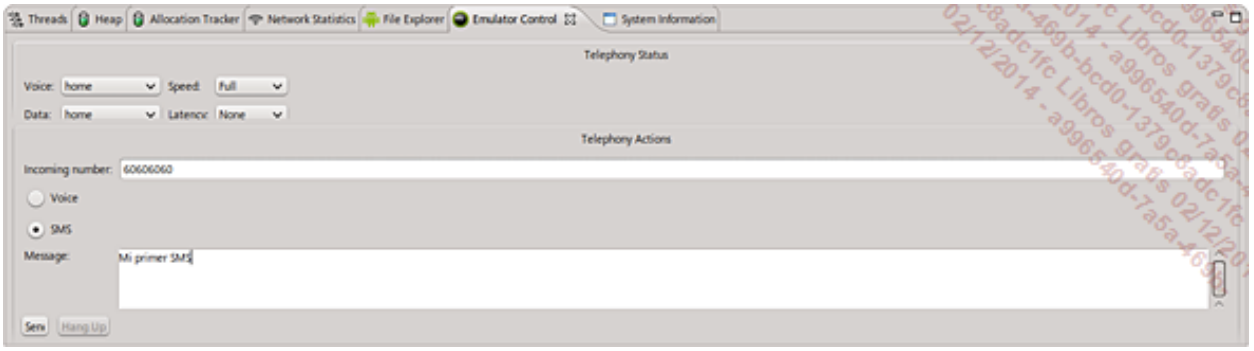
```
ArrayList<String> devidedMessageList =
smsManager.divideMessage(body);
smsManager.sendMultipartTextMessage(receiver, null,
devidedMessageList, null, null);
```

En este caso, utilice el método **sendMultipartTextMessage** para enviar el mensaje.

Este método recibe los siguientes parámetros:

- Una cadena de caracteres que representa el destinatario del mensaje.
- Una cadena de caracteres que representa el servicio de envío de mensajes que se utilizará. Puede pasar **null** para utilizar el servicio de envío por defecto.
- Una lista de cadenas de caracteres que se corresponden con las distintas partes del mensaje dividido.
- Un **PendingIntent** que se ejecutará si el envío del mensaje finaliza (con éxito o error). Puede pasar **null** para ignorar el evento de envío del mensaje.
- Un **PendingIntent** que se ejecutará cuando el destinatario reciba el mensaje (con éxito o error). Puede pasar **null** para ignorar el evento de recepción del mensaje.

Puede simular el envío de mensajes mediante la herramienta **DDMS** (sólo funciona con el emulador).



2. Recepción de mensajes

Una aplicación puede reaccionar ante la recepción de un mensaje mediante un Broadcast Receiver.

```
<receiver android:name="MySMSReceiver">
<intent-filter>
   <action android:name="android.provider.Telephony.SMS_RECEIVED" />
</intent-filter>
</receiver>
```

La aplicación debe, sin embargo, declarar la *permission* **RECEIVE_SMS**.

```
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
```

Una vez declarado el receiver, debe implementar la clase que gestiona la recepción de un mensaje.

```
public class MySMSReceiver extends BroadcastReceiver {

   public final String RECEIVE_MSG =
"android.provider.Telephony.SMS_RECEIVED";

@Override
public void onReceive(Context context, Intent intent) {
```

```
    if (intent.getAction().equals(RECEIVE_MSG)) {
         Bundle extra = intent.getExtras();
         if (extra != null) {
             Object[] pdus = (Object[]) extra.get("pdus");
             final SmsMessage[] messages = new
SmsMessage[pdus.length];
             for (int i = 0; i < pdus.length; i++) {
                  messages[i] =
SmsMessage.createFromPdu((byte[]) pdus[i]);
             }
             if (messages.length > -1) {
             for (int i = 0; i < messages.length; ++i) {
                  final String messageBody =
messages[i].getMessageBody();
                  final String phoneNumber =
messages[i].getDisplayOriginatingAddress();
                  Toast.makeText(context, "Emisor: " +
phoneNumber,  Toast.LENGTH_LONG).show();
                  Toast.makeText(context, "Mensaje: " +
messageBody,Toast.LENGTH_LONG).show();
                      }
                  }
             }
         }
    }
}
```

En el método **onReceive** hay que comprobar, en primer lugar, que el evento recibido corresponde a la recepción de un mensaje.

La lectura del mensaje se realiza siguiendo los siguientes pasos:

- El mensaje recibido se obtiene de los extras del intent.
- A continuación, se obtiene la lista de mensajes recibidos en forma de tabla. Para extraerlos, se utilizan los **pdu** (*Protocol Data Unit*). Este protocolo sirve para encapsular los datos de un mensaje y facilitar su transferencia.
- Después, se obtienen los mensajes a partir de la tabla inicializada anteriormente.
- Una vez se han creado los mensajes, se recorren y se muestra su contenido.

Cámara

1. Utilizar la aplicación Cámara del dispositivo

La mayoría de los dispositivos Android tienen una o varias cámaras que puede utilizar en su aplicación para permitir al usuario capturar fotos o grabar vídeos. Cada dispositivo que disponga de una cámara tendrá una aplicación que permita grabar fotos y vídeos.

El objetivo es iniciar la aplicación de fotos y obtener el resultado, lo que corresponde a una llamada al método **startActivityForResult**.

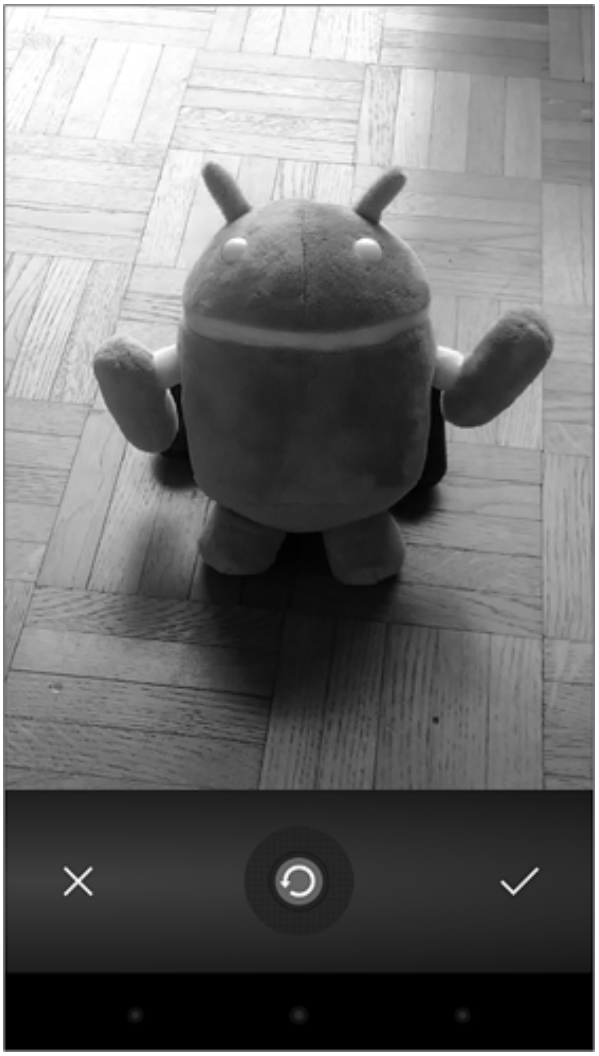
El intent utilizado puede tener como acción:

- **MediaStore.ACTION_IMAGE_CAPTURE**: útil para capturar una foto.
- **MediaStore.ACTION_VIDEO_CAPTURE**: útil para crear un vídeo.

Por lo tanto, puede utilizar las siguientes líneas de código para llamar a la aplicación de fotos del dispositivo.

```
startActivityForResult(new Intent(MediaStore.ACTION_IMAGE_CAPTURE),
IMAGE_CAPTURE);
```

Lo que dará:



Una vez se ha capturado la foto, el usuario la podrá validar, volver a capturar o cancelar la captura de fotos.

- La validación desencadenará un **setResult(RESULT_OK)**.
- La anulación desencadenará un **setResult(RESULT_CANCELLED)**.
- La imagen se incluirá en los extras (identificador **data**) en forma de vista previa.

La obtención de la vista previa de la imagen se realiza del siguiente modo:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
   super.onActivityResult(requestCode, resultCode, data);
   if (requestCode == IMAGE_CAPTURE) {
      if (data != null) {
         Bitmap picture =
data.getParcelableExtra("data");
         if (picture != null) {
         //Realizar el tratamiento con la imagen
         }
      }
   }
}
```

Este método le permite obtener únicamente la vista previa de la imagen. Si desea obtener la foto completa, debe modificar las características específicas del intent usado en la llamada al método**startActivityForResult**.

El objetivo es especificar la URI del archivo usado para almacenar la imagen y añadirla a los extras del intent (clave **MediaStore.EXTRA_OUTPUT**).

Si su aplicación requiere la presencia de una cámara en el dispositivo, debe especificar esta restricción en el manifiesto para que la aplicación no pueda iniciarse en dispositivos que no tengan cámara.

```
<uses-feature android:name="android.hardware.camera" />
```

También puede comprobar dinámicamente si un dispositivo tiene o no cámara.

```
if
(context.getPackageManager().hasSystemFeature(PackageManager.
FEATURE_CAMERA)){
        // Este dispositivo tiene cámara
} else {
        // Este dispositivo no tiene cámara
}
```

2. Controlar la cámara

También puede tomar, directamente, el control de la cámara, configurarla, registrar un vídeo o capturar una imagen sin pasar por otra aplicación. Este método requiere una *permission*.

```
<uses-permission android:name="android.permission.CAMERA"/>
```

Para el uso de la cámara, hay dos pasos que son imprescindibles:

- Comenzar a usar la cámara:

```
Camera camera = Camera.open();
```

- Finalizar el uso de la cámara:

```
camera.release();
```

Una cámara tiene varios parámetros modificables (focus, flash, zoom…), son accesibles mediante la clase **Parameters** (subclase de **Camera**).

```
Camera.Parameters param = camera.getParameters();
```

3. Grabar un vídeo

También puede grabar vídeos de dos formas distintas:

- Mediante un intent (véase la sección Utilizar la aplicación Cámara del dispositivo).
- Mediante la API de Cámara.

Además de la *permission* **CAMERA** declarada anteriormente, necesitará una nueva *permission*:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

La grabación de vídeo requiere el uso de la clase **MediaRecorder**. Ésta permite especificar varias características de la grabación de vídeo:

- Fuente de audio y vídeo,
- Formato del archivo de salida,

- Tipo de encoding para el audio y el vídeo, etc.

Para realizar una grabación, debe seguir los siguientes pasos:

Crear una instancia de la clase **Camera**, utilizando el método **open**.

```
Camera camera = Camera.open();
```

- Crear una instancia de la clase **MediaRecorder**.

```
MediaRecorder mediaRecorder = new MediaRecorder();
```

- Asociar la instancia de **MediaRecorder** con la instancia de la cámara.

```
camera.unlock();
mediaRecorder.setCamera(camera);
```

- Especificar las particularidades del **MediaRecorder** (fuente de audio y vídeo, archivo de salida...).

```
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
mediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.
QUALITY_HIGH));
mediaRecorder.setOutputFile(getOutputMediaFile(MEDIA_TYPE_VIDEO).
toString());
```

- Especificar el espacio usado para mostrar la vista previa del vídeo.

```
mediaRecorder.setPreviewDisplay(preview.getHolder().getSurface());
```

- Preparar el **MediaRecorder** para la grabación.

```
mediaRecorder.prepare();
```

- Comenzar la grabación.

```
mediaRecorder.start();
```

- Acabar la grabación y liberar el **MediaRecorder** y la cámara.

```
mMediaRecorder.release();
camera.release();
```

Sensores en Android

1. Principio

Un dispositivo Android dispone de un gran número de sensores (acelerómetro, giroscopio, sensor de proximidad...) y todos ellos pueden gestionarse mediante la clase **SensorManager**.

Esta clase se comporta del mismo modo que los diferentes **Manager** disponibles en Android (**NotificationManager**, **NetworkManager**...). Debe utilizar el método **getSystemService** para inicializar el **SensorManager**.

```
final SensorManager sensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
```

Hay varios tipos de sensores. A continuación se muestra una lista exhaustiva:

- **Sensor.TYPE_ACCELEROMETER**: representa el acelerómetro (dirección actual en los tres ejes en m/s^2).
- **Sensor.TYPE_GYROSCOPE**: representa el giroscopio del dispositivo, que permite conocer el porcentaje de rotación del dispositivo en los tres ejes.
- **Sensor.TYPE_PROXIMITY**: representa un sensor de proximidad que sirve para determinar la distancia entre el dispositivo y un objeto determinado. Se utiliza, en particular, para detener el dispositivo cuando se está comunicando.

Puede obtener la lista de sensores disponibles de un dispositivo mediante el método **getSensorList**.

```
final SensorManager sensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);

List<Sensor> sensorsList =
sensorManager.getSensorList(Sensor.TYPE_ALL);

for (Sensor sensor : sensorsList) {
   Log.v("SensorActivity", "sensor = " + sensor.getName());
}
```

Obtendrá el siguiente resultado en un Galaxy Nexus, por ejemplo:

```
sensor = Rotation Vector Sensor
sensor = Gravity Sensor
sensor = Linear Acceleration Sensor
sensor = Orientation Sensor
sensor = Corrected Gyroscope Sensor
```

También puede obtener la lista de los sensores que corresponden a un tipo de sensor (por ejemplo, para los sensores de proximidad).

```
List<Sensor> proximitySensorsList =
sensorManager.getSensorList(Sensor.TYPE_PROXIMITY);
```

También puede averiguar el sensor (por defecto) que gestiona un tipo de sensores.

```
Sensor defaultProximitySensor =
sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
Log.v("SensorActivity", "defaultProximitySensor = " +
```

```
defaultProximitySensor.getName());
```

Lo que dará, por ejemplo, para un Galaxy Nexus:

```
defaultProximitySensor = GP2A Proximity sensor
```

2. Acelerómetro

El acelerómetro es un sensor muy práctico que sirve, en particular, para conocer la aceleración del dispositivo o los movimientos realizados. Es especialmente útil en los videojuegos que utilizan los movimientos del dispositivo como método de control.

Los movimientos se pueden detectar según tres ejes:

- **Eje x** (de izquierda a derecha y viceversa).
- **Eje y** (de alante a atrás y viceversa).
- **Eje z** (de arriba a abajo y viceversa).

Para poder utilizar el acelerómetro y recibir las actualizaciones de los distintos movimientos del dispositivo, tiene que suscribirse a las actualizaciones mediante un **SensorEventListener**.

Debe seguir los siguientes pasos:

- Obtener una instancia del **SensorManager**.
- Declarar e implementar una instancia de la clase **SensorEventListener**.
- Asociar el **SensorManager** al listener.

Para ilustrar esta funcionalidad, cree un proyecto Android que muestre la evolución de los valores del acelerómetro en los tres ejes (usando **TextView**).

Para comenzar, hay que crear una interfaz compuesta de tres **TextView**.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/x_axis"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="10dp"
        android:textSize="30sp" />

    <TextView
        android:id="@+id/y_axis"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="10dp"
        android:textSize="30sp" />

    <TextView
        android:id="@+id/z_axis"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
```

```
        android:layout_marginTop="10dp"
        android:textSize="30sp" />

</LinearLayout>
```

- Ahora, inicialice los distintos elementos que componen su interfaz en el método **onCreate**.

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.accelerometer);

   xTextView = (TextView) findViewById(R.id.x_axis);
   yTextView = (TextView) findViewById(R.id.y_axis);
   zTextView = (TextView) findViewById(R.id.z_axis);
}
```

- A continuación, recupere la instancia de la clase **SensorManager**.

```
final SensorManager sensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
```

- Cree una instancia de la clase **SensorEventListener**.

```
private final SensorEventListener accelerometerEventListener =  new
SensorEventListener() {

   @Override
   public void onSensorChanged(SensorEvent event) {
   if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
      xAxis = event.values[0];
      yAxis = event.values[1];
      zAxis = event.values[2];
      xTextView.setText("Axe x = " + Float.toString(xAxis));
      yTextView.setText("Axe y = " + Float.toString(yAxis));
      zTextView.setText("Axe z = " + Float.toString(zAxis));
   }
   }

   @Override
   public void onAccuracyChanged(Sensor sensor, int accuracy) {

   }
};
```

Debe sobrecargar los siguientes dos métodos:

- **onSensorChanged**: permite advertir las actualizaciones del sensor.
- **onAccuracyChanged**: permite advertir los cambios de precisión del sensor.

La implementación del método **onSensorChanged** se realiza siguiendo los siguientes pasos:

- Compruebe que el tipo de sensor afectado por estas modificaciones se corresponde con un acelerómetro.
- Recupere los nuevos valores de posición del acelerómetro en un orden predefinido (x, y y z).
- Actualice los valores de los campos de texto.

- Para finalizar, asocie el acelerómetro al **SensorManager** registrando el listener.

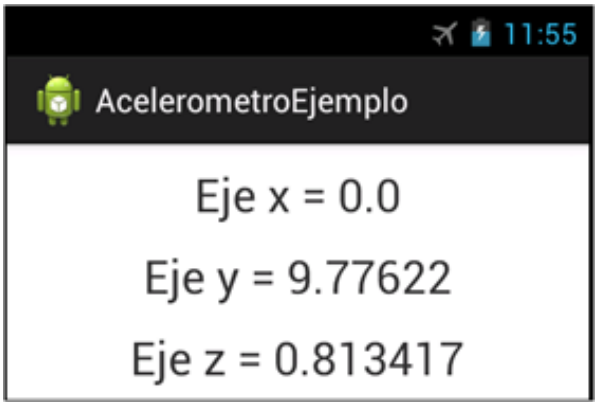
```
sensorManager.registerListener(accelerometerEventListener,
```

```
sensorManager.getDefaultSensor(sensorType),
SensorManager.SENSOR_DELAY_NORMAL);
```

Para ello, utilice el método **registerListener** que recibe tres parámetros:

- El listener que se registrará.
- El sensor utilizado por el dispositivo que se corresponde con el tipo acelerómetro.
- El intervalo de actualización del sensor.

Lo que dará:



- Si la diferencia entre el valor anterior y el nuevo valor del eje X es positiva, significa que el movimiento se realiza hacia la derecha.
- Si la diferencia entre el valor anterior y el nuevo valor del eje Y es positiva, significa que el movimiento se ha efectuado hacia arriba.
- Si la diferencia entre el valor anterior y el nuevo valor del eje Z es positiva, significa que el movimiento se ha realizado hacia adelante.

3. Giroscopio

El giroscopio es un sensor que permite averiguar el ángulo de inclinación y la velocidad angular del dispositivo.

El uso del giroscopio es idéntico al del acelerómetro. Por lo tanto, hay que:

- Recuperar una instancia del **SensorManager**.
- Declarar e implementar una instancia de la clase **SensorEventListener**.
- Asociar el **SensorManager** al listener.

Lo que se traduce en pequeñas modificaciones en relación al ejemplo anterior:

```
   private int sensorType = Sensor.TYPE_GYROSCOPE;
  private final SensorEventListener gyroscopeEventListener = new
SensorEventListener() {
   @Override
   public void onSensorChanged(SensorEvent event) {
      if (event.sensor.getType() == Sensor.TYPE_GYROSCOPE) {
         xAxis = event.values[0];
         yAxis = event.values[1];
         zAxis = event.values[2];
         xTextView.setText("Axe x = " + Float.toString(xAxis));
         yTextView.setText("Axe y = " + Float.toString(yAxis));
         zTextView.setText("Axe z = " + Float.toString(zAxis));
      }
```

```
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {

    }
};
```

4. Sensor magnético

El sensor magnético permite averiguar el valor del campo magnético que se aplica a su dispositivo (en microteslas).

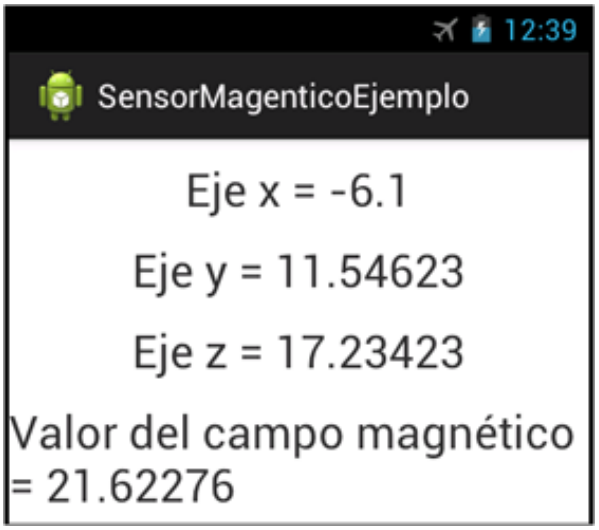
La forma de utilizar el sensor magnético es idéntica a la de los sensores descritos anteriormente:

- Recuperar una instancia del **SensorManager**.
- Declarar e implementar una instancia de la clase **SensorEventListener**.
- Asociar el **SensorManager** al listener.

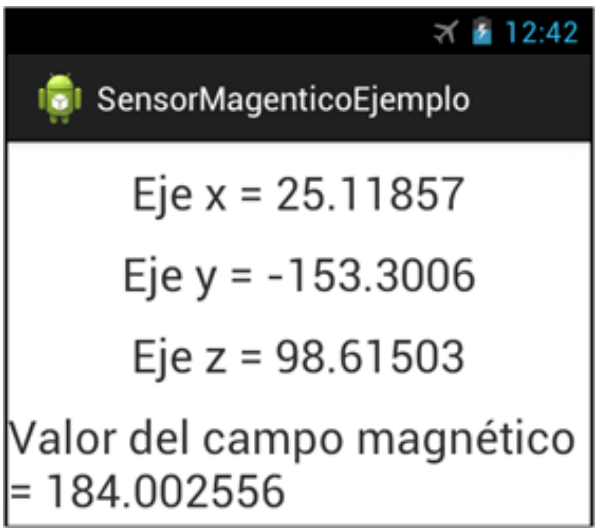
Se va a retomar el ejemplo anterior añadiendo un campo de texto que sirva para mostrar el valor del campo magnético que captura el dispositivo.

```
private final SensorEventListener magneticEventListener = new
SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {
        xAxis = event.values[0];
        yAxis = event.values[1];
        zAxis = event.values[2];
        magneticValues = Math.sqrt((double) (xAxis * xAxis
            + yAxis * yAxis + zAxis * zAxis));
        xTextView.setText("Axe x = " + Float.toString(xAxis));
        yTextView.setText("Axe y = " + Float.toString(yAxis));
        zTextView.setText("Axe z = " + Float.toString(zAxis));
        magneticValueTextView.setText("Valor del campo
magnético = " + magneticValues);
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {

    }
};
```

Lo que dará:



Si intenta realizar la prueba de aproximar un imán a su dispositivo, podrá observar un aumento significativo del valor del campo magnético que detecta el dispositivo.



Bluetooth

La mayoría de los dispositivos Android tienen **Bluetooth** y el framework Android ofrece APIs que sirven para facilitar el uso del Bluetooth en sus aplicaciones. Puede utilizar esta API para intercambiar datos, conectarse a otros dispositivos que soportan esta tecnología, etc.

La API Bluetooth le permite:

- Activar el Bluetooth si lo necesita.
- Escanear y asociar los dispositivos entre ellos.
- Transferir los datos.

El primer paso necesario e importante en el uso de las APIs Bluetooth consiste en comprobar si el dispositivo en cuestión efectivamente tiene Bluetooth.

La clase **BluetoothAdapter** le permite saber rápidamente si un dispositivo dispone o no de la tecnología Bluetooth. Esto es así gracias al método **getDefaultAdapter**.

```
BluetoothAdapter bluetoothAdapter =
BluetoothAdapter.getDefaultAdapter();
if (bluetoothAdapter != null) {
   Toast.makeText(this, "Este dispositivo tiene
Bluetooth", Toast.LENGTH_LONG).show();
} else {
   Toast.makeText(this, "Este dispositivo no tiene
Bluetooth", Toast.LENGTH_LONG).show();
}
```

La inicialización del **BluetoothAdapter** puede devolver dos valores:

- **null**, si el dispositivo no dispone de la tecnología Bluetooth.
- En caso contrario, una instancia de la tecnología Bluetooth.

No olvide añadir la *permission* que permite acceder a las APIs Bluetooth.

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

1. Activar el Bluetooth

El **Bluetooth** puede estar disponible en el dispositivo pero inactivo. Puede activarlo de dos formas:

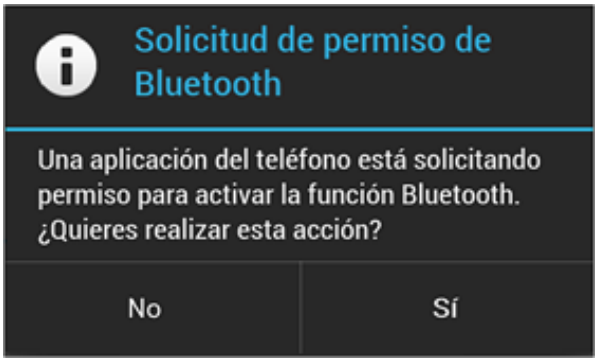
- Solicitar al usuario si desea activar el Bluetooth.
- Activar directamente el Bluetooth.

El primer método es el recomendado, ya que solicita al usuario su autorización antes de activarlo. Para ello, basta con llamar al método **startActivityForResult** pasando como parámetro el Intent**BluetoothAdapter.ACTION_REQUEST_ENABLE**.

```
if(!bluetoothAdapter.isEnabled()) {
   startActivityForResult(new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE),
BLUETOOTH_ACTIVATION);
}
```

No olvide comprobar si el Bluetooth está activado mediante el método **isEnabled**.

Lo que dará:



El usuario tendrá la elección de activar o no el Bluetooth y el resultado de esta decisión se podrá obtener en el método **onActivityResult**.

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
   super.onActivityResult(requestCode, resultCode, data);
   if (requestCode == BLUETOOTH_ACTIVATION) {
      if (resultCode == RESULT_OK) {
         Toast.makeText(BluetoothActivity.this,
"El usuario ha activado el Bluetooth",
Toast.LENGTH_LONG).show();
      } else {
         Toast.makeText(BluetoothActivity.this,
"El usuario no ha activado el Bluetooth",
Toast.LENGTH_LONG).show();
      }
   }
}
```

También puede forzar la activación del Bluetooth con el método **enable**.

```
bluetoothAdapter.enable();
```

Esta técnica fuerza la activación del Bluetooth sin solicitar la autorización del usuario y requiere otra*permission* (la que sirve para administrar el Bluetooth del dispositivo).

```
<uses-permission
android:name="android.permission.BLUETOOTH_ADMIN"/>
```

2. Permitir a otros dispositivos conectarse

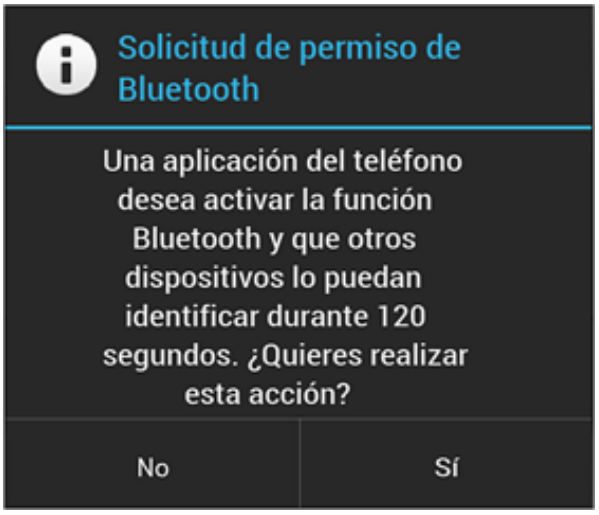
Una aplicación puede hacer que el dispositivo sea detectable y, de este modo, permitir a otros dispositivos conectarse al mismo.

Para ello, debe utilizar el método **startActivityForResult** con un Intent cuyo valor sea**ACTION_REQUEST_DISCOVERABLE**.

```
startActivityForResult(new
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE), BLUETOOTH_SCAN);
```

Podrá recuperar el resultado en el método **onActivityResult**.

Lo que dará:



Puede conocer la lista de dispositivos ya asociados al dispositivo en cuestión a través del método**getBondedDevices**. A continuación, puede recorrer fácilmente la lista de dispositivos que se han encontrado.

```
Set<BluetoothDevice> knownDevices =
bluetoothAdapter.getBondedDevices();
for (BluetoothDevice device: knownDevices) {
   Log.v("BluetoothActivity", "dispositivo = " + device.getName());
}
```

La búsqueda de nuevos dispositivos es un tratamiento asíncrono y se realiza mediante Broadcast Receivers. Puede suscribirse a dos Broadcast Receivers, el que gestiona el inicio del proceso de búsqueda de dispositivos y el invocado cuando la búsqueda ha finalizado.

```
private BroadcastReceiver discoverDevicesStarted = new
BroadcastReceiver() {

   @Override
   public void onReceive(Context context, Intent intent) {
   if (intent.getAction().equals(ACTION_DISCOVER_START)) {
      //Comienzo de la búsqueda de nuevos dispositivos
   } else if (intent.getAction().equals(ACTION_DISCOVER_END)) {
       //Final de la búsqueda de nuevos dispositivos
   }
};
};
```

Debe registrar los Broadcast Receivers en el método **onResume** de su actividad. No olvide cancelar la suscripción en el método **onPause**.

```
registerReceiver(discoverDevicesStarted, new
IntentFilter(ACTION_DISCOVER_START));
registerReceiver(discoverDevicesStarted, new
IntentFilter(ACTION_DISCOVER_END));
```

Para iniciar la búsqueda de nuevos dispositivos, utilice el método **startDiscovery**.

```
bluetoothAdapter.startDiscovery();
```

El método **cancelDiscovery** permite anular y detener la búsqueda de nuevos dispositivos.

Lo que dará:

```
@Override
protected void onResume() {
   super.onResume();
   registerReceiver(discoverDevicesStarted, new
IntentFilter(ACTION_DISCOVER_START));
   registerReceiver(discoverDevicesStarted, new
IntentFilter(ACTION_DISCOVER_END));
bluetoothAdapter.startDiscovery();
}

@Override
protected void onPause() {
   super.onPause();
   unregisterReceiver(discoverDevicesStarted);
   bluetoothAdapter.cancelDiscovery();
}
```

Ahora, puede obtener la lista de dispositivos encontrados en el **Broadcast Receiver** y, en concreto, gracias a la acción **ACTION_FOUND**. Esta acción se invoca cada vez que se encuentra un nuevo dispositivo.

```
private BroadcastReceiver discoverDevicesStarted = new
BroadcastReceiver() {

   @Override
   public void onReceive(Context context, Intent intent) {
      if (intent.getAction().equals(ACTION_DISCOVER_START)) {
         //Comienzo de la búsqueda de nuevos dispositivos
   } else if (intent.getAction().equals(ACTION_DISCOVER_END)) {
         //Final de la búsqueda de nuevos dispositivos
   } else if
(BluetoothDevice.ACTION_FOUND.equals(intent.getAction())) {
      if (intent != null &&
intent.hasExtra(BluetoothDevice.EXTRA_DEVICE)) {
      BluetoothDevice newDevice =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            }
         }
      };
   };
```

El comportamiento del Bluetooth, para la conexión y la transferencia de datos, es similar al comportamiento de un cliente y un servidor usando un socket para conectarse.

Puede utilizar las siguientes dos clases para establecer conexiones:

- **BluetoothServerSocket**: utilizada en un servidor para atender conexiones que vienen de diferentes clientes.
- **BluetoothSocket**: utilizada por el cliente para establecer una conexión a un servidor.

La transferencia de datos se realiza a través de **InputStream.read** y **OutputStream.write**.

NFC

Los últimos dispositivos Android disponen de la tecnología NFC (*Near Field Communication* - Comunicación de campo cercano). Esta tecnología permite transferir datos entre dispositivos cuando éstos se tocan.

Esta tecnología podría tener muchas utilidades:

- Medio de pago,
- Tarjeta de transporte, etc.

Para utilizar la tecnología NFC en un dispositivo Android hay que declarar la siguiente *permission*:

```
<uses-permission android:name="android.permission.NFC"/>
```

Si su aplicación debe leer un tag NFC cuando contacte con otro dispositivo, la actividad que se ocupa de la lectura del tag NFC debe tener:

- La acción **NDEF_DISCOVERED**.
- La categoría **DEFAULT**.
- Una etiqueta **data** que sirve para especificar el tipo de dato que se leerá.

Lo que da:

```
<activity android:name=".NFCActivity">
<intent-filter>
  <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="http" android:host="www.ediciones-eni.com "/>
</intent-filter>
</activity>
```

Esta actividad se utilizará como actividad por defecto para todos los tags NFC que apuntan a un sitio web **http** y, en particular, con las direcciones que apunten al sitio especificado en el atributo **host**.

Extraer datos de un tag NFC es muy sencillo. Cuando utiliza su aplicación como lector de tag NFC, se pasa un Intent a su actividad con los datos alojados en el tag.

```
final String nfcAction = NfcAdapter.ACTION_NDEF_DISCOVERED;
String action = getIntent().getAction();

if (action.equals(nfcAction)) {
   Parcelable[] nfcTagMsg =
getIntent().getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);

   for (Parcelable nfcTag : nfcTagMsg) {
      NdefMessage msg = (NdefMessage) nfcTag;
      NdefRecord[] records = msg.getRecords();

      for (NdefRecord record : records) {
      //realizar los tratamientos en los diferentes campos
que componen el tag NFC
      }
   }
}
```

Recupere la tabla de datos correspondiente a los mensajes transmitidos por NFC.

Recorra estos datos y recupere los mensajes almacenados.

A continuación, recupere los distintos campos que componen cada mensaje y realice el tratamiento deseado.

1. Android Beam

La versión 4.0 de Android incluye una nueva funcionalidad que sirve para transmitir datos (enlaces, contactos, etc.) mediante NFC, se trata de Android Beam. Puede utilizar directamente esta funcionalidad para usar NFC en su aplicación.

Como se ha descrito anteriormente, necesitará la *permission* NFC.

Para recibir datos mediante Android Beam, hay que modificar el manifiesto y, en particular, la actividad que permite recibir datos a través de NFC.

```
<activity android:name=".NFCActivity" >
<intent-filter>
 <action android:name="android.nfc.action.NDEF_DISCOVERED" />
 <category android:name="android.intent.category.DEFAULT" />
 <data android:mimeType="application/com.eni.android.beam" />
</intent-filter>
</activity>
```

Observe que el atributo tipo del elemento data apunta a su aplicación, la cual gestiona los tags (enviados por la aplicación Android Beam).

A continuación, basta con recuperar los datos como se ha descrito anteriormente (**getParcelableArrayExtra**, **getRecords**, etc.).

Para enviar mensajes mediante Beam debe utilizar el siguiente método:

```
setNdefPushMessageCallback(NfcAdapter.CreateNdefMessageCallback
callback, Activity activity)
```

Este método permite enviar un mensaje en modo push mediante Android Beam. Recibe como parámetros un método que sirve para crear el mensaje así como una instancia de la actividad relacionada con el mensaje. Lo que dará:

```
NfcAdapter nfcAdapter = NfcAdapter.getDefaultAdapter(this);

nfcAdapter.setNdefPushMessageCallback(new
CreateNdefMessageCallback() {
   @Override
   public NdefMessage createNdefMessage(NfcEvent event) {
      //Cree su NdefMessage
      return null;
   }
}, this);
```

Existe otro método disponible que permite recibir el equivalente a un acuse de recibo del mensaje NFC.

```
setOnNdefPushCompleteCallback(NfcAdapter.OnNdefPushCompleteCallback
callback, Activity activity)
```

TTS (Text To Speech)

El TTS es una API de Android que permite al Framework Android leer un texto determinado. Para ilustrarlo, cree un proyecto Android compuesto por un campo de texto de edición y un botón. Se leerá el texto que se encuentra en el campo de texto.

- Cree una instancia de la clase **TextToSpeech** mediante su constructor. Este último recibirá como parámetro el contexto así como una instancia de la clase **OnInitListener** (subclase de**TextToSpeech**).

```
private TextToSpeech tts;
...
tts = new TextToSpeech(this, ttsListener);
```

- A continuación, cree una instancia de la clase **OnInitListener** y sobrecargue el método **onInit**, que le permite inicializar las propiedades de su TTS (idioma usado, etc.).

```
private TextToSpeech.OnInitListener ttsListener =
new OnInitListener() {
   @Override
   public void onInit(int status) {
      if (status == TextToSpeech.SUCCESS) {
         tts.setLanguage(Locale.getDefault());
      }
   }
};
```

El código anterior permite comprobar el estado del TTS e inicializar el idioma.

La variable **status** puede tener el valor **SUCCESS** o **ERROR** especificando, de este modo, la disponibilidad o no del **TextToSpeech**.

El método **setLanguage** permite especificar el idioma utilizado por el TTS.

El método **Locale.getDefault()** permite obtener el idioma por defecto del dispositivo.

Cuando el usuario haga clic en el botón, debe obtener el texto introducido para iniciar la lectura mediante el método **speak**.

```
ttsButton.setOnClickListener(new OnClickListener() {
   @Override
   public void onClick(View v) {
      String text = ttsEditText.getText().toString();
      tts.speak(text, TextToSpeech.QUEUE_FLUSH, null);
   }
});
```

El método **speak** recibe tres parámetros:

- El texto que se leerá.
- La gestión de la cola de lectura de mensajes: este argumento puede tener uno de los dos siguientes valores:
  - **QUEUE_ADD** permite añadir un mensaje en la cola de lectura.
  - **QUEUE_FLUSH** permite vaciar la cola de lectura y leer los mensajes que están en espera.
- El tercer parámetro permite especificar la clase utilizada para leer mensajes. Este parámetro es opcional, y puede pasar **null** como valor para utilizar la clase por defecto.

- Para finalizar, no olvide liberar el TTS en el método **onStop**.

```
@Override
protected void onStop() {
    super.onStop();
    if (tts != null) {
        tts.stop();
        tts.shutdown();
    }
}
```